NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

**FAULT TOLERANT COMPUTING TESTBED:
A TOOL FOR THE ANALYSIS OF HARDWARE AND
SOFTWARE FAULT HANDLING TECHNIQUES**

by

John C. Payne, Jr.

December 1998

Thesis Advisor:	Alan Ross
Second Reader:	Douglas J. Fouts

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 1998	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE
FAULT TOLERANT COMPUTING TESTBED: A TOOL FOR THE ANALYSIS OF HARDWARE AND SOFTWARE FAULT HANDLING TECHNIQUES

5. FUNDING NUMBERS

6. AUTHOR(S)
Payne, John C. Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Operating computers in space requires the use of very expensive radiation hardened microelectronics devices. Unfortunately, the United States radiation hardened market is rapidly shrinking and makes up a very small percentage of the commercial market. For these reasons, and the fact that commercial-off-the-shelf (COTS) devices are cheaper, more capable, readily available, and software availability is much greater, the use of COTS devices in future space systems is fast becoming a reality. A significant disadvantage of COTS devices is their susceptibility to radiation induced single event upsets (SEUs), among other radiation effects which are detrimental to electronic systems.

This thesis focuses on the board level design of a tool which enables the analysis of fault tolerant computing techniques in a laboratory environment in the presence of radiation induced SEUs. When implemented, this tool will be beneficial to the study of using COTS devices in space. The tool will provide the capability to analyze the performance of hardware redundancy techniques and software algorithms intended to improve the performance of COTS microprocessors in this environment prior to their use in designs intended for actual space applications. Cadence Concept™ design schematics, associated Verilog® code and simulation results are presented to develop this concept.

14. SUBJECT TERMS
Fault Tolerant Computing, Triple Modular Redundancy (TMR), Commercial-off-the-shelf (COTS) Devices, Single Event Upsets (SEUs), Cadence Concept Schematic, Verilog

15. NUMBER OF PAGES 184

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

FAULT TOLERANT COMPUTING TESTBED: A TOOL FOR THE ANALYSIS OF HARDWARE AND SOFTWARE FAULT HANDLING TECHNIQUES

John C. Payne, Jr.
Lieutenant, United States Navy
B.S., Virginia Polytechnic Institute and State University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1998

Author: John C. Payne. Jr.
John C. Payne, Jr.

Approved by: Alan a Ross
Alan Ross, Thesis Advisor

Douglas J. Fouts
Douglas J. Fouts, Second Reader

[illegible]
Jeffrey B. Knorr, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

Operating computers in space requires the use of very expensive radiation hardened microelectronics devices. Unfortunately, the United States radiation hardened market is rapidly shrinking and makes up a very small percentage of the commercial market. For these reasons, and the fact that commercial-off-the-shelf (COTS) devices are cheaper, more capable, readily available, and software availability is much greater, the use of COTS devices in future space systems is fast becoming a reality. A significant disadvantage of COTS devices is their susceptibility to radiation induced single event upsets (SEUs), among other radiation effects which are detrimental to electronic systems.

This thesis focuses on the board level design of a tool which enables the analysis of fault tolerant computing techniques in a laboratory environment in the presence of radiation induced SEUs. When implemented, this tool will be beneficial to the study of using COTS devices in space. The tool will provide the capability to analyze the performance of hardware redundancy techniques and software algorithms intended to improve the performance of COTS microprocessors in this environment prior to their use in designs intended for actual space applications. Cadence Concept™ design schematics, associated Verilog® code and simulation results are presented to develop this concept.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

The author would like to thank Mr. Kenny Clark and Mr. Mark Hall for their assistance in obtaining information about the triple-vote R3081 experiment which was part of the Microelectronics and Photonics Testbed.

The author wants to thank Professor Ross for his guidance and patience during the work in performing this investigation.

I. INTRODUCTION

A. BACKGROUND

A fault tolerant system is one that can continue the correct performance of its specified tasks in the presence of hardware and/or software faults. Fault tolerance is the attribute that enables a system to achieve fault tolerant operation. In many sensitive applications fault tolerant computing techniques are employed where the failure of these systems could lead to disastrous results. Examples of such sensitive applications include aircraft and spacecraft flight control systems and power plant control systems. A recent example of such a failure occurred with the loss of PanAmSat's Galaxy 4 satellite.

> Galaxy 4's attitude control system and an identical backup unit conked out at approximately 6 p.m. Eastern Daylight Time May 19 [1998], sending the satellite into an uncontrolled spin. [Ref. 1]

While the loss of this satellite was not necessarily "disastrous," it could indeed prove to be very expensive. The Galaxy 4 cost between $200 million and $250 million to build, launch, and insure. [Ref. 1]

In the space environment there are three categories of radiation effects in integrated circuits. Total Dose

Effects, Dose Rate Effects, and Single Event Effects. Within Single Event Effects are the four sub-categories: Single Event Upset (SEU), Single Event Latchup (SEL), Single Event Gate Rupture (SEGR), and Single Event Burnout (SEB). Total Dose Effects and Dose Rate Effects are destructive effects in integrated circuits arising from solar flares, neutrons from nuclear detonations, and protons in the Van Allen belts. In addition, three of the subcategories of single event effects (SEL, SEGR, and SEB) are also destructive. These effects must be compensated for with the use of radiation hardening and shielding techniques. On the other hand, SEUs, which are essentially bit flips occurring within a device due to ionized charge being collected in a circuit, can be reduced by hardware architecture and software techniques such as redundancy.

Operating computers in the space environment requires the use of very expensive radiation hardened (rad-hard) devices. In addition to the use of rad-hard technology, space systems also employ many other approaches to fault tolerance such as hardware redundancy, fault tolerant software algorithms, error detecting/correcting codes, etc. While deploying reliable, fault tolerant computers in space will always require rad-hard components, the number of suppliers of such devices is decreasing and the costs of the devices continues to increase. Many manufacturers are abandoning their production of rad-hard devices in favor of the more lucrative, booming consumer electronics industry. According to the May 1997 issue of *Military & Space Electronics*, "U.S. Department of Defense (DOD) leaders are struggling to find new ways to safeguard the dwindling supplier base of radiation-hardened microelectronics that are necessary to meet future spacecraft requirements." [Ref. 2]

While the commercial satellite industry may fill the void, it is estimated that DOD must increase investments from $30 million per year to nearly $60 million per year to advance the technology and ensure a base of reliable suppliers. [Ref. 2]

The issue is in the fabrication process of the microelectronic devices. The production of the unique rad-hard devices requires specialized processes and demand for them is considerably less than that for consumer electronics. With the costs of modern fabrication lines reaching nearly $2.8 billion apiece, it is obviously cost prohibitive for companies to merely have two separate production facilities: one for rad-hard devices and one for non-rad-hard devices. A company producing both rad-hard and non-rad-hard devices will have to give up precious fabrication time to make a few devices for a limited market. This precious time takes away from the production of microelectronics for a booming PC market and could mean millions, if not billions, of dollars in lost revenue. Herein lies the fundamental economic reason for the escalating prices of rad-hard microelectronics.

An approach to solving this problem, which is receiving considerable amount of research, is the development of new processes that allow companies to manufacture rad-hard devices without major changes to their fabrication process. Another possible approach is the development of alternative approaches in hardware and software fault tolerant design with non-rad-hard commercial-off-the-shelf (COTS) microelectronics to reduce the dependency on rad-hard technology. This research project addresses the latter approach.

B. PURPOSE

The goal of this research is to develop a fault tolerant computing testbed for use as a tool for the analysis of hardware and software fault handling techniques. In particular, the testbed is intended to allow the analysis of techniques to resolve faults caused by single event upsets. The testbed computer will employ a three CPU, triple modular redundant (TMR), design. The TMR testbed will allow flexibility in the hardware and software design enabling direct performance analysis of various approaches to fault tolerant design. The testbed will enable fault injection simulations and direct radiation testing on the system for data analysis and hardware/software benchmarking.

This project will help in the development of cheaper alternatives to the highly expensive radiation hardened devices. It will further the research of radiation testing and single-event upset research by providing a testbed for analysis of various hardware redundancy techniques as well as any software techniques chosen to be employed. The testbed will be used in direct radiation testing in a laboratory environment and/or placed in a satellite as an experimental payload to study the effects in the actual flux environment of the satellite. This study will benefit our development of small, economical satellites for both commercial and military use.

C. THESIS ORGANIZATION

The organization of this thesis largely follows the approach taken to the design of a TMR system. Chapter I is a brief introduction with background information. Chapter II describes the microprocessor selection process and the characteristics of the selected processor. Chapter III presents various topics in hardware redundancy including triple modular redundancy, voting techniques, synchronization and timing issues. Chapter IV contains the actual hardware design of the testbed. Simulation and results are presented in Chapter V. Finally the conclusions drawn from this research are presented in Chapter VI.

II. PROCESSOR SELECTION

A. CHARACTERISTICS

The place to start when designing a computer is with processor selection. The selection of the processor, or processors in the case of hardware redundancy, is where critical decisions are made regarding expected operating environment, necessary performance, power consumption and space limitations.

1. COTS vs. Rad-Hard

In June 1994, a directive was issued by then Secretary of Defense William Perry requiring the use of COTS parts in military systems whenever possible. As previously discussed, the availability of rad-hard parts is diminishing and as a result military, NASA, and commercial spacecraft builders may eventually be forced to use COTS technology.

There are significant advantages to using COTS devices. COTS devices tend to be state-of-the-art and are therefore significantly more capable than rad-hard devices. To put it in perspective, often the choice is between a COTS Pentium or a rad-hard 286 or 386 microprocessor. As an example, in July 1998 Space Electronics announced intentions to release a single-board computer for space designed with primarily

COTS devices. This product, running at 66 MHz, is intended to compete with the RAD6000 from Lockheed Martin Federal Systems, which runs at 33 MHz and costs twice as much. [Ref. 3] The processor used in the new release product, the 6U VME SB486R radiation hardened 32-bit single board computer based on Intel's 80486 microprocessor, is still an order of magnitude slower than the 300-400 MHz microprocessors currently available for desktop PCs.

Other advantages of COTS systems include lower cost and better availability. Often a rad-hard microprocessor can cost many (10-15) thousands of dollars more than more capable, current technology COTS devices. In addition, rad-hard devices often have uncertain delivery times. Because of the declining rad-hard device market, these devices often must be special ordered from a limited number of available manufacturers. On the other hand, manufacturers of COTS devices often have stockpiles and can deliver a product within 24-48 hours. Many powerful COTS devices can even be obtained over the counter at several big name electronics stores.

Commercial software is much more available for COTS devices. Software development is a very costly part of building any computer system. As the complexity of microprocessors increases, so does the complexity of the required software. If rad-hard devices are not identical to their COTS counterparts, software must be specially designed for this device. This is both expensive and time consuming. In addition, this specially designed software will have to undergo rigorous testing to check its response to unexpected situations. [Ref. 4] This is in contrast with software for COTS devices where large companies design software for these devices. The software becomes proven over time through the high volume of users and the consumers actually participate in the testing of these products.

Finally, while not necessarily an advantage of COTS devices themselves, it is possible to achieve some degree of radiation hardness by employing various techniques to shield COTS devices which are not themselves radiation hardened. [Ref. 4] While the use of shielding has shown to improve the reliability of devices in radiation environments, it adds to the physical space and weight requirements.

However, there are disadvantages to using COTS devices. While the reliability of COTS devices used in benign environments is known, their reliability in stressing environments (radiation, thermal, vibration) is uncertain. [Ref. 4] The susceptibility of COTS devices to radiation induced failures is a major concern, and survivability in the space environment may be difficult with many COTS devices. While some COTS devices may have hardness levels of 100 kRADs or more, this hardness varies greatly from one device to another. This hardness varies even for devices produced by the same manufacturer. Because of this lack of hardness assurance by manufacturers, each individual device will have to undergo testing and effectively be space qualified.

Another disadvantage of COTS devices is they change rapidly. The semiconductor industry generally cycles new technology every 6-18 months. The devices continue to get faster, more capable, and require less power. The advantage here is clear for devices intended for the normal, non-stressing environment. However, as the devices get smaller, faster, and more complex, they are becoming more susceptible to radiation. Finally, in many cases, the required safety and reliability specifications, especially for military applications, simply cannot be met by COTS devices. [Ref. 4]

2. CISC vs. RISC

Reduced instruction set computer (RISC) machines were designed to take advantage of the caching, prefetching, pipelining, and superscalar methods that were invented to improve the performance of complex instruction set computer (CISC) machines. The CISC machines depend on long complex instructions. The operand access for these instructions required complex address arithmetic. As a result, CISC machines were unable to take full advantage of these techniques.

The RISC focuses on reducing the number and complexity of instructions in the machine. This allows a reduction of actual machine hardware complexity. Early on, RISC machines operated such that each instruction completed in one clock cycle. This was achieved by limiting the instructions in RISC machines to a fixed length, usually 1 word. Thus, in a 32-bit machine, one 32-bit word specifies everything there is to know about the instruction.

With the advent of pipelining, the current goal is that (at least) one instruction will begin and (at least) one instruction will complete during every clock cycle. Since program execution time depends on throughput and not on individual instruction execution times, issuing (and thus completing) one instruction per clock cycle is an appropriate goal. This is achieved by making instructions simple, not by making the clock period longer.

3. Size, Pinout, Power

The size of the device determines the physical space required on the assembled board. Space and weight constraints are critical limitations imposed on systems for satellites and other space applications. Similarly, power consumption is a critical factor in space applications where a steady, endless supply of power from a standard 120 volt outlet is not available. In applications where power comes from batteries and/or solar cells, available power is a precious commodity.

The pinout of the device is often directly related to its physical size. In addition, many devices reduce their pinout requirements by having individual address and data lines multiplexed together on one interface pin.

4. Bus Width and Memory Size

The bus width of COTS devices essentially follows current trends. While many processors are available today with 64-bit architectures, the RAD6000 microprocessor (considered to be the industry standard for radiation hardened microprocessors) incorporates a 32-bit architecture. Compared to 32-bit architectures, a 64-bit bus effectively doubles (design dependent) the pinout requirements and correspondingly increases the power consumption of the device.

As bus size increases, the complexity of the interconnectivity hardware increases as well. Particularly in a TMR design where 3 microprocessors are connected together with voting hardware, increasing the bus width from

32-bits to 64-bits requires a rather significant increase in hardware and logic.

The size of the physical memory that the processor can use is a significant factor in space applications as well. In space applications where large volume secondary storage media is generally not available, the bigger the physical memory potential the better. Of course, this is essentially limited by the bus architecture of the device. A device with a 64-bit bus can accommodate a larger physical memory space than a 32-bit bus device. Without large secondary storage media, all operations will be performed using ROM and RAM with varying combinations of ROM and RAM types depending on the application. Therefore, it is necessary that the available physical address (memory) space be large enough to accomplish the intended tasks.

5. Speed

The speed of the device is an important issue. However, in a TMR design, the speed at which the system can operate will be limited by the propagation time of the voting and vote error control logic as well as the memory setup and hold times. Although new personal computers are currently available with processors running at 300-400 MHz, the current new radiation hardened microprocessors run at 33-66 MHz.

The speed of the microprocessor chosen for this TMR design will be limited by the critical path logic propagation time in the several FPGAs chosen to implement the voting and vote error control.

6. Multiple Chip vs. Single Chip Implementations

The tradeoff associated with a single chip processor versus a processor which requires additional hardware peripheral devices is a significant issue. This is especially true in a TMR design where each address/data line as well as each control line has to be voted to ensure agreement between the three processors. In addition, in space applications the potential for radiation induced error increases with each additional piece of hardware added. Other problems include fault localization. With microprocessors with external peripheral device requirements, voting and vote error control complexity is increased. Also, board reliability is inversely proportional to the number of chips on it.

The overall complexity of the board design increases as well with microprocessors with external peripheral device requirements. In a TMR design, this increased complexity is compounded. In a single chip microprocessor, the associated interface complexity is internal to the device. Therefore radiation-induced faults are limited to a single device when performing processor voting which corresponds to simpler voting logic and less hardware requirements.

B. PROCESSOR REVIEW

As part of this research, several microprocessors were analyzed based on the microprocessor characteristics discussed in the preceding section. Tables 1, 2, and 3 contain data concerning the various COTS CISC and RISC microprocessors that were considered in developing the testbed.

The processor chosen was the R3081 RISC Microcontroller manufactured by Integrated Device Technologies (IDT). The reasons for this selection were many. From the outset of this research project, the intent was to choose a COTS device for the TMR design.

The R3081 is a COTS, single chip, RISC architecture machine, with a 32-bit multiplexed address/data bus. The highly flexible and user configurable device can run between 20 and 50 MHz and is readily available.

The determining factor for selecting the R3081 was the availability of radiation environment performance data from the Naval Research Laboratory (NRL). The R3081 was used in a triple vote experiment deployed on the Microelectronics and Photonics Testbed (MPTB). The MPTB is a space experiment launched in 1997 into a high radiation orbit to test performance, reliability, and survivability of new microelectronics and photonic devices operating in the space radiation environment. The triple vote experiment was one of 24 experiments onboard the MPTB which were individually scheduled by a core controller. The purpose of this experiment was to measure SEU, SEL, and Total Dose effects in IDT R3081 microprocessors vs. epi thickness. The three microprocessors used had epi thicknesses of 6, 8, and 12 microns respectively. The MPTB design was obtained from NRL and used as a starting point for the testbed designed in this research project.

Characteristic	AM29000	AM29050	PowerPC 603e
Manufacturer	AMD	AMD	IBM, Motorola
Processor Architecture	Streamlined Instruction	Streamlined Instruction	RISC
Package	168-PQFP or 169-PGA	169-PGA	
Floating Point Accelerator	Y (off chip)	Y	Y
Memory Management Unit	Y	Y	Y
Speed (MHz)	16-33	20-40	200-250
Integer Multiply/Divide	Y	N	Y
Bus Architecture	32-bit 3 bus	32-bit 3 bus	Selectable 64-/32-bit data bus, 32-bit address bus
Demultiplex Signal	N/A	N/A	N/A
Physical Address Space			
Power (watts)	< 1	< 1	3.5 - 5.8
Single Chip	N	Y	Y
Built-in Master/Slave	Y	Y	Y

Table 1. Microprocessor Review (1 of 3).

Characteristic	PowerPC 604e	PowerPC 750	R3081
Manufacturer	IBM, Motorola	IBM, Motorola	IDT
Processor Architecture	RISC	RISC	MIPS/RISC
Package	255-CBGA	360-CBGA	84-pin MQUAD/PLCC
Floating Point Accelerator	Y	Y	Y
Memory Management Unit	Y	Y	Y
Speed (MHz)	250-350	200-300	20-50
Integer Multiply/Divide	Y	Y (3)	Y (2)
Bus Architecture	64-bit data, 32-bit address	32-bit data, 64-bit address	32-bit address/data multiplexed
Demultiplex Signal	N/A	N/A	Y
Physical Address Space			4GB
Power (watts)	6.0-14.5	4.7-11.0	2.375-4.125
Single Chip	Y	Y	Y
Built-in Master/Slave	N	N	N

Table 2. Microprocessor Review (2 of 3).

Characteristic	R36100	R4650	R5000
Manufacturer	IDT	IDT	IDT
Processor Architecture	MIPS/RISC	MIPS-III/RISC	MIPS-IV/RISC
Package	208-pin MQUAD	288-pin MQUAD	223-pin CPGA or 272-ball SBGA
Floating Point Accelerator	N	Y	Y
Memory Management Unit	Y	Y	Y
Speed (MHz)	20-33	100-180	200
Integer Multiply/Divide	Y	Y	Y
Bus Architecture	8-, 16-, 32-bit programmable address and data	32- or 64-bit address/data multiplexed	64-bit address/data multiplexed
Demultiplex Signal	N/A	Y	Y
Physical Address Space	4GB	4GB	
Power (watts)	2-3	1.646-3.465	7.59-8.25
Single Chip	Y	Y	Y
Built-in Master/Slave	N	N	N

Table 3. Microprocessor Review (3 of 3).

C. CHARACTERISTICS OF SELECTED PROCESSOR

The IDT R30xx family of microprocessors is intended to offer the high-performance associated with the MIPS RISC architecture for low-cost, simplified, power sensitive applications. [Ref. 5] Some features of the R3081E include:

- High level of integration minimizes cost
- Over 40 MIPS at 50 MHz
- Low cost 84-pin packaging
- Large on-chip user configurable instruction and data caches
- On chip Floating Point Accelerator (FPA)
- 20 through 50 MHz operation
- Multiplexed address/data bus interface with low cost, low speed memory systems with high speed CPU support
- On-chip 4-deep write buffer eliminates memory write stalls
- On-chip 4-deep read buffer supports burst or simple block reads

Figure 1 shows a block diagram of the IDT R3081E microprocessor. Some of the highlights include:

- System Control Coprocessor (CP0)
  - ✓ Dedicated Exception/Control Registers
  - ✓ Dedicated Memory Management Registers
- Integer CPU Core
  - ✓ 32 32-bit general registers
  - ✓ ALU, Shifter, Mult/Div Unit, Address Adder, and PC Control
- Floating Point Coprocessor (CP1)
  - ✓ 16 64 bit registers
  - ✓ Exponent, Add, Divide, and Multiply Units
  - ✓ Floating Point Exception/Control
- Configurable Instruction and Data Caches
- 4-deep Read and Write Buffers

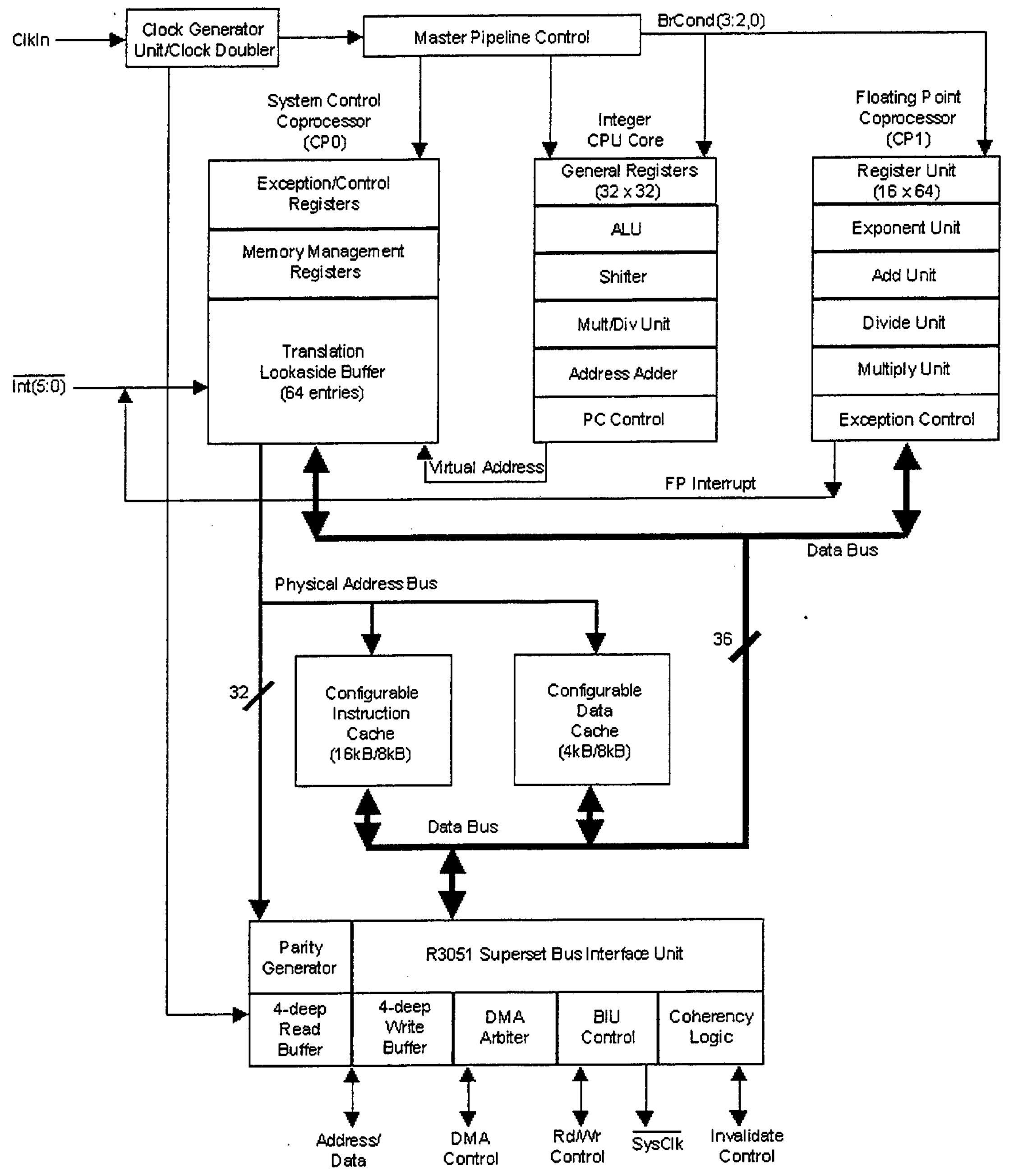


Figure 1. IDT R3081 Block Diagram. From Ref. [5].

1. CPU Core

The CPU Core is a full 32-bit RISC integer execution engine, capable of sustaining close to a single cycle per instruction rate. It contains a 5 stage pipeline and 32 orthogonal 32-bit registers. [Ref. 5]

2. System Control Co-Processor

The integrated on-chip System Control Co-Processor (CP0) manages both the exception handling of the CPU and the virtual to physical address mapping. The fully associative 64-entry Translation Lookaside Buffer (TLB) maps 4kB virtual pages into the physical address space. The virtual to physical mapping includes kernel segments which are hard-mapped to physical addresses, and kernel and user segments which the TLB maps 4kB page by 4kB page into anywhere in the 4GB (potentially) physical address space. The TLB also allows 8 pages to be locked by the kernel to ensure deterministic response in real-time applications. [Ref. 5]

3. Floating Point Co-Processor

The R3081 also incorporates an integrated R3010A compatible FPA which is co-processor 1 (CP1) to the CPU. The high-performance co-processor provides separate add, multiply, and divide functional units for single and double precision floating point arithmetic. To the software engineer, the FPA simply appears as an extension of the integer execution unit with 16 dedicated 64-bit floating point registers. The software references these as 32 32-bit registers when performing loads or stores. [Ref. 5]

4. Clock Generator Unit

The on-chip clock generator manages the interaction of the CPU core, caches, and bus interface. It includes a clock doubler to provide a higher frequency signal to the internal execution core. [Ref. 5]

5. Instruction and Data Caches

The on-chip cache is default configured to 16kB Instruction Cache and 4kB Data Cache. However, the cache can be reconfigured by system software to 8kB of Instruction and 8kB of Data caches. The instruction cache is organized with a line size of 16 bytes (four 32-bit entries) which achieves hit rates in excess of 98% in most applications. The data cache is organized as a line size of 4 bytes (one word) and achieves hit rates near 95% in most applications. The high hit rates associated with the instruction and data cache contribute significantly to the performance of the R3081E. The instruction cache is a direct mapped cache capable of caching instructions from anywhere in the 4GB physical address space. The instruction cache is implemented using physical addresses and physical tags (rather than virtual addresses or tags) to eliminate the requirement of flushing on context switch. As with the instruction cache, the data cache is a direct mapped physical address cache capable of mapping any word within the 4GB physical address space. However, the data cache is implemented as a write-through cache to insure that main memory is always consistent with cache memory. In order to minimize processor stalls due to data write operations, the bus interface utilizes a 4-deep write buffer which "captures" address and data information at the processor execution rate, allowing it to be written to main memory at the memory speeds with minimum impact to overall system performance. [Ref. 5]

6. Bus Interface Unit

Because the R3081 uses its large internal caches to provide the majority of the bandwidth requirements of the execution engine, it can utilize a much simpler bus interface connection to slower memory. The bus interface utilizes a 32-bit address and data bus multiplexed onto a single set of pins. It also provides an ALE (Address Latch Enable) output signal to de-multiplex the A/D bus, and simple handshaking signals to process CPU read and write requests. The DMA Arbiter allows an external master to control the external bus if desired. As described previously in the Instruction and Data Cache section, a 4-deep write buffer decouples the speed of the execution engine from the speed of the main memory system. The write buffers capture and FIFO processor address and data information in store operations and schedule them on the bus at a rate that can be handled by the system memory. The read interface is capable of both single word and quad word reads. Single word reads utilize a simple handshake, and quad word reads can utilize either a simple handshake or a tighter timing mode when the memory system can burst data at the processor clock rate. In order to accommodate slower quad word reads, the 4-deep read buffer FIFO is utilized allowing the external interface to queue data within the processor before releasing it to perform a "burst" fill of the internal caches. [Ref. 5]

7. System Usage

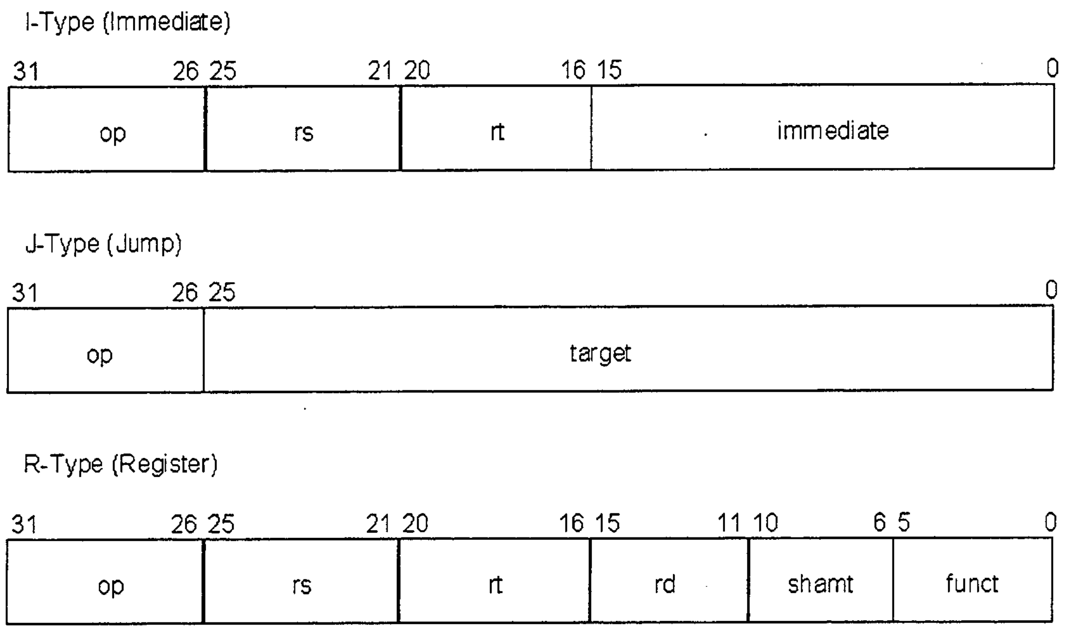
The bus interface of the IDT R30xx (including the R3081E) family was specifically designed to allow a wide range of memory systems. A typical system using off-the-shelf logic devices contains simple transparent latches to de-multiplex the R30xx address and data busses and the A/D bus; the data path between the memory system and the A/D bus is managed by octal transceivers; and a small set of PALs is used to control the various data path elements, and to control the handshake between the memory and the processor. [Ref. 5]

8. Instruction Set Architecture

All instructions and addresses are 32 bits and the CPU utilizes a 5-stage pipeline to achieve a near one instruction per clock cycle execution rate. There are five basic groups of instructions:

- Load/Store
  - ✓ Move data between memory and general registers
- Computational
  - ✓ Perform arithmetic, logical, and shift operations on values in registers
- Jump and Branch
  - ✓ Change control flow of program
- Co-Processor
  - ✓ Perform operations on the co-processor set
- Special
  - ✓ Movement of data between special and general registers, system calls, breakpoint operations

Figure 2 displays the instruction formats of the R3081 processor. Load/Store instructions are all encoded as Immediate, or I-Type, instructions. Computational instructions are encoded as either Register, or R-Type, instructions when both source operands and the result are general registers or I-Type when one of the source operands is a 16-bit immediate value. Jump and Branch instructions can be either J-Type (target address is PC + 26-bit immediate value), R-Type (target address is 32-bit value contained in one of general registers), or I-Type (Branch Instructions where target address is formed from a 16-bit displacement relative to the PC). Jump and Link instructions save a return address in register R31. Co-processor Loads and Stores are always I-Type. Special instructions are always encoded as R-Type. [Ref. 5]



where:

op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target register or branch condition
immediate	16-bit immediate, or branch or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
shamt	5-bit shift amount
funct	6-bit function field

Figure 2. Instruction Formats. After Ref. [5].

Table 4 lists the instruction set mnemonics of R3081E processor.

9. The Pipeline Architecture

The execution of a single instruction is performed in five separate steps:

- Instruction Fetch (IF)
  - ✓ Instruction virtual address translated to physical address and read from internal instruction cache
- Read (RD)
  - ✓ Instruction decoded and required operands read
- ALU (ALU)
  - ✓ Required operation is performed
- Memory Access (MEM)
  - ✓ If instruction was a Load or Store, the data cache is accessed
- Write Back (WB)
  - ✓ Results from ALU step updated in on-chip register file

Figure 3 illustrates the pipeline and the capability to execute 5 instructions per cycle. Pipeline hazards in the

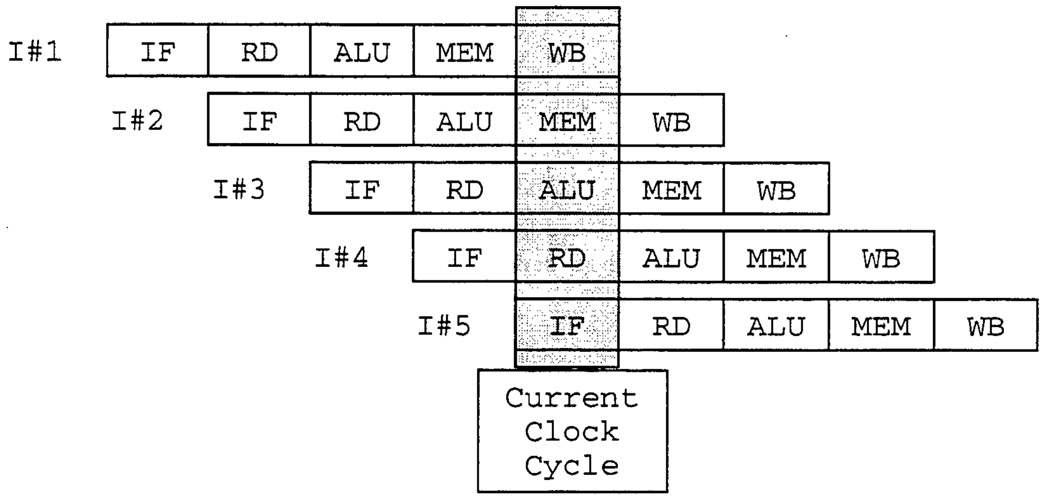


Figure 3. 5-Instructions per Clock Cycle. After Ref. [5].

R3081 are handled in both hardware and software. The hardware methods used are forwarding and stalling (minimal). The hardware methods deal with instructions that need a result from the register file of the immediately prior instruction and in integer multiply and divide operations where an instruction attempts to access the LO or HI registers prior to completion of the multiply or divide. If this happens, the requesting instruction will be blocked until the result is ready. The software method used is an optimizing compiler and peephole scheduler of the assembler. Two instruction classes which use the software method are Load instructions and Jump and Branch instructions. Both of these instruction classes have a delay, or latency, of one cycle. Rather than include extensive pipeline control logic, the CPU gives responsibility for dealing with "delay slots" to software. The peephole optimizer, performed as a part of compilation or assembly, can reorder the code to insure that the instruction in the delay slot does not require the logical result of the "delayed" instruction. [Ref. 5]

D. SUMMARY

Having completed a review of some of the desired characteristics of a microprocessor to be investigated when designing a system, the IDT R3081 RISC microprocessor was

OP	DESCRIPTION	OP	DESCRIPTION
	Load/Store Instructions		**Multiply/Divide Instructions**
LB	Load Byte	MULT	Multiply
LBU	Load Byte Unsigned	MULTU	Multiply Unsigned
LH	Load Halfword	DIV	Divide
LHU	Load Halfword Unsigned	DIVU	Divide Unsigned
LW	Load Word		
LWL	Load Word Left	MFHI	Move From HI
LWR	Load Word Right	MTHI	Move To HI
SB	Store Byte	MFLO	Move From LO
SH	Store Halfword	MTLO	Move To LO
SW	Store Word		
SWL	Store Word Left		**Jump and Branch Instructions**
SWR	Store Word Right	J	Jump
		JAL	Jump and Link
	Arithmetic Instructions	JR	Jump to Register
	(ALU Immediate)		
ADDI	Add Immediate	JALR	Jump and Link Register
ADDIU	Add Immediate Unsigned	BEQ	Branch on Equal
SLTI	Set on Less Than Immediate	BNE	Branch on Not Equal
SLTIU	Set on Less Than Immediate Unsigned	BLEZ	Branch on Less Than or Equal to Zero
ANDI	AND Immediate	BGTZ	Branch on Greater Than Zero
ORI	OR Immediate	BLTZ	Branch on Less Than Zero
XORI	Exclusive OR Immediate	BGEZ	Branch on Greater Than or Equal to Zero
LUI	Load Upper Immediate	BLTZAL	Branch on Less Than Zero and Link
		BGEZAL	Branch on Greater Than or Equal to Zero and Link
	Arithmetic Instructions		**Special Operations**
	(3-operand, register type)		
ADD	Add	SYSCALL	System Call
ADDU	Add Unsigned	BREAK	Break
SUB	Subtract		
SUBU	Subtract Unsigned		**Coprocessor Instructions**
SLT	Set on Less Than	LWCz	Load Word from Coprocessor
SLTU	Set on Less Than Unsigned	SWCz	Store Word to Coprocessor
AND	AND	MTCz	Move to Coprocessor
OR	OR	MFCz	Move from Coprocessor
XOR	Exclusive OR	CTCz	Move Control to Coprocessor
NOR	NOR	CFCz	Move Control from Coprocessor
		COPz	Coprocessor Operation
	Shift Instructions	BCzT	Branch on Coprocessor z True
SLL	Shift Left Logical	BCzF	Branch on Coprocessor z False
SRL	Shift Right Logical		
SRA	Shift Right Arithmetic		**System Control Coprocessor (CP0)**
			Instructions
SLLV	Shift Left Logical Variable	MTC0	Move to CP0
SRLV	Shift Right Logical Variable	MFC0	Move from CP0
SRAV	Shift Right Arithmetic Variable	TLBR	Read Indexed TLB Entry
		TLBWI	Write Indexed TLB Entry
		TLBWR	Write Random TLB Entry
		TLBP	Probe TLB for Matching Entry
		RFE	Restore from Exception

Table 4. Instruction Mnemonics. After Ref. [5].

chosen. Although the performance of the R3081 is much less than that of the current microprocessors available, it does have the performance and computing power necessary for analyzing fault tolerant improvement techniques in the presence of radiation induced SEUs. In addition, the R3081 has previously been tested by the Naval Research Laboratory and flown in actual space satellite experiments. Finally, the R3081 employs a flexible bus interface which makes it a good candidate for use in a redundant hardware design.

In the next chapter, some of the concepts of triple modular redundancy, a hardware redundancy technique, are covered. This is followed by a description of a simple R3081 based system. Finally, a brief overview of how three R3081 processors were incorporated into a redundant design is presented.

III. HARDWARE REDUNDANCY

There are many techniques available to achieve some degree of fault tolerance. Fault tolerant systems basically employ some combination of hardware, software, time, or information redundancy. The purpose of this chapter is to introduce the concept of triple modular redundancy (TMR). The initial design of the testbed will employ a TMR design and as such TMR issues are dealt with thoroughly. The overall goal of this project is to design a testbed which allows flexibility in the ultimate methods employed to achieve fault tolerance. This will allow the user to compare and contrast the fault tolerant performance of many combinations of the different techniques.

A. TRIPLE MODULAR REDUNDANCY (TMR)

A common form of hardware redundancy is triple modular redundancy (TMR). The basic concept is fairly simple. It involves the triplication of the hardware and performing a majority vote to determine the output of the system. This technique is considered to be passive hardware redundancy in that it masks the occurrence of faults. Fault tolerance is achieved through the use of majority voting techniques without the need for fault detection or system recovery. [Ref. 6] If one of the modules becomes faulty, the two remaining modules, which are fault-free, mask the fault when the majority vote is performed. In short, TMR uses three identical modules, performing identical operations, with a majority voter determining the output, as shown in Figure 4.

In a TMR system with three microprocessors, an SEU could cause one processor to branch to a completely wrong address. That processor will continue to cause errors on all votes until it is reset to the state of the correct processors. Until it is reset, the system is no longer a TMR system. It is a dual processor with comparison system which provides for error detection but no error correction.

One of the primary disadvantages with a TMR system is that the system can be no more reliable than the voter itself. Indeed the voter becomes a single point of failure—if the voter fails, the entire system fails. [Ref. 6] Several techniques can be used to overcome this. One method is the use of triplicated voters which produce three independent outputs. Here again three identical modules receive identical inputs and perform the same operations on those inputs. Each module provides its output to three separate and independent voters to produce the three results, as shown in Figure 5. Each output is correct as long as no more than one module, or input, is faulty. In essence, the voter is no longer the single point of failure.

A multi-stage TMR system can be built by interconnecting this triplicated voter approach as shown in Figure 6. [Ref. 6] A multi-stage system with triplicated voters can provide some error correction in that an error in a module of one stage is masked and the voters provide three

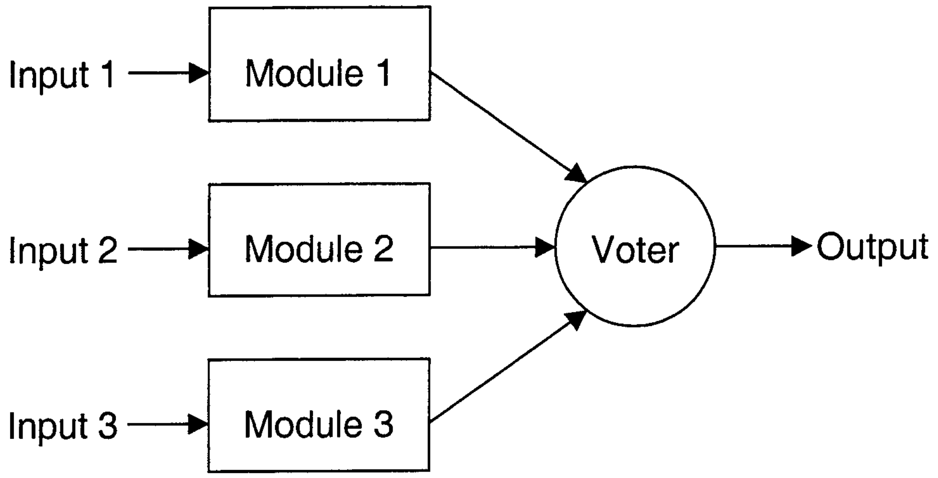


Figure 4. Triple Modular Redundancy. After Ref. [6].

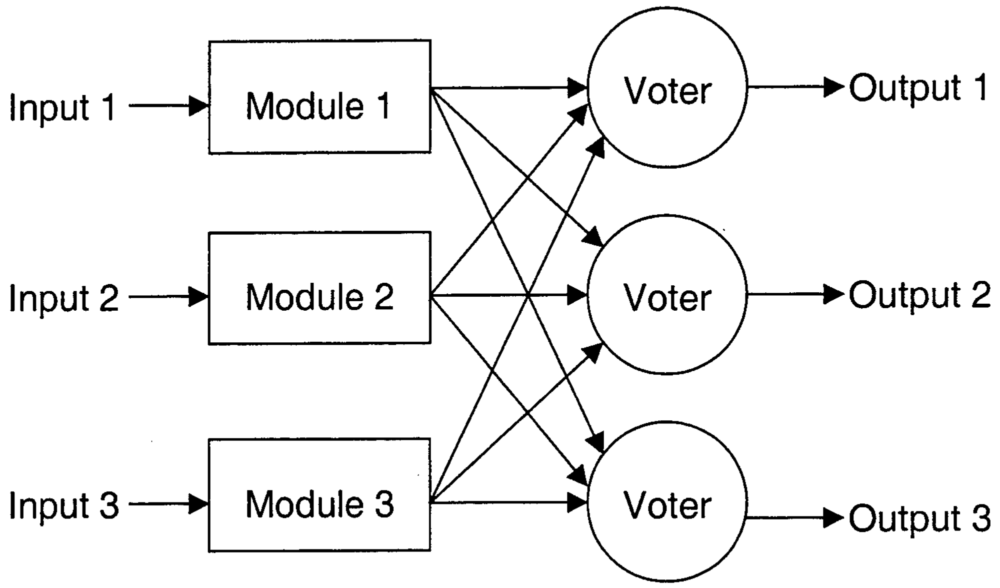


Figure 5. TMR with triplicated voters. After Ref. [6].

independent and "corrected" results to the next stage. At the final stage, the three independent outputs can then be voted again to form a single output. However, this final voter could again become the single point of failure.

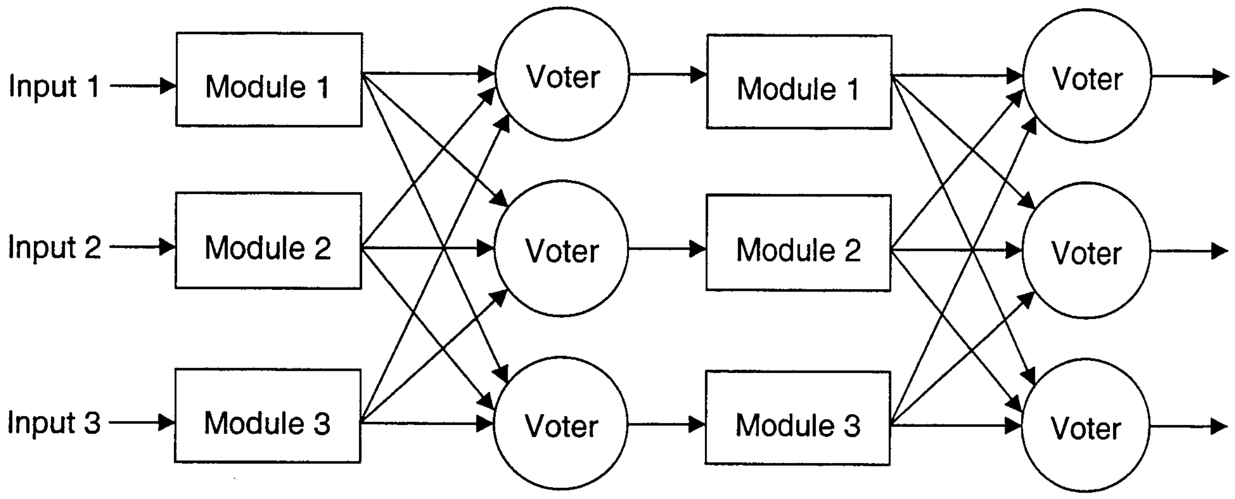


Figure 6. Multiple-stage TMR system. After Ref. [6].

A generalization of the TMR approach is *N-Modular Redundancy (NMR)*. [Ref. 6] TMR is based on the techniques of NMR. There are *N* redundant modules vice three. In general, *N* is chosen to be odd so that majority voting techniques can still be applied. The advantage gained is that more module faults may be tolerated. In an NMR system with *N* redundant modules, majority voting will allow the system to tolerate faults in $\lceil N/2 \rceil - 1$ modules. The primary concerns associated with NMR system deal with added logic hardware and circuit complexity. Clearly, one could design a system that continues to employ NMR voting at multiple stages to improve system reliability. Referring to Figure 5, the triplicated voters could even be voted again to ensure faults are detected in the voters themselves. This could conceivably continue in an endless cycle. Practical applications and design constraints often prevail and are the limiting factor to choosing *N* in an NMR system. [Ref. 6]

1. Voting Techniques

Voting may occur at several locations within a system. Take, for example, a TMR system used as an industrial process controller. [Ref. 6] The controller could sample from three identical, independent sensors and perform a vote to determine which sensor value to use. This data is provided to three identical, independent modules to perform some calculations on the sensor data, and then a majority vote on these calculations is performed to perhaps adjust the controls of the process. The voting can be used on both analog and digital data. This approach masks and contains the effect of a faulty sensor. An alternative method might be to provide the values from each of the three sensors directly to a dedicated module, perform the necessary calculations, and then vote the results from the three modules. Here, faulty sensor data would be allowed to migrate into the processing modules. The tradeoffs between the two approaches are slight but would obviously have to be analyzed to determine the appropriate design based on the application.

A hardware voter is a relatively simple circuit to design and implement. All that is needed is a combinational logic circuit that produces a 1 when a majority of the input bits are 1 and a 0 when a majority of the input bits are 0. An implementation of a one-bit majority voter is shown in Figure 7. Alternately, the carry out output of a 1-bit full adder will produce the necessary output to implement the 1-bit majority voter. An 8, 16, 32, or 64-bit voter can be constructed by replicating the circuit in Figure 7 in

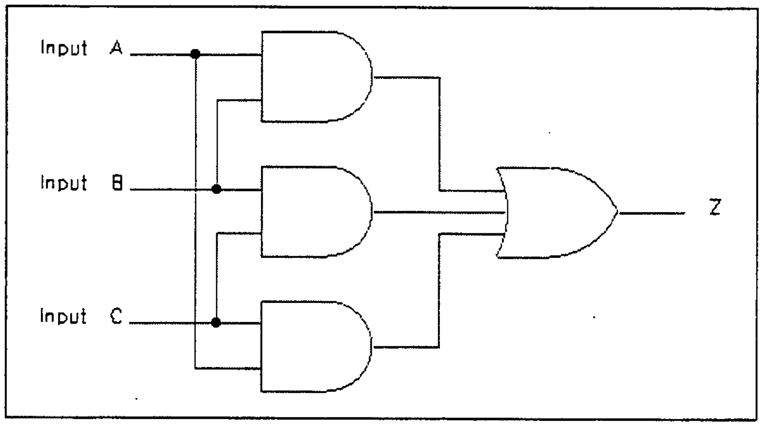


Figure 7. 1-bit majority voter. After Ref. [6].

parallel for each bit that needs to be voted. One can see the amount of additional logic grows rapidly if, for instance, the three independent modules in a TMR system to be voted are 32-bit microprocessors. The desired reliability will certainly have to be weighed against the space, power, and weight limitations, especially in satellite and other space applications.

2. Voting Issues

In practical applications, timing will have to be considered when performing majority voting. If the three inputs to a majority voter arrive at different times, then depending on when the output of the voter is sampled, an incorrect vote may be generated. In many applications, an incorrect result cannot be allowed even for a very small period of time. [Ref. 6] There are techniques which can be applied that will force the inputs to the voter to be synchronized so that the output of the voter is sampled at the correct time. One approach to achieving synchronization involves a two-phase clock which drives master-slave D flip-flops on each input to the majority voter. The costs of using this synchronization approach will be in terms of additional logic and timing delays.

Another problem that may be encountered in hardware voting is that the three modules in a TMR system, or the three sensors that feed the three modules, could disagree slightly even in a fault-free environment. These devices, sensors in particular, can seldom be produced so that they generate identical results under the same circumstances. In addition, a single analog-to-digital converter can produce results that differ slightly in the least significant bits, even if the exact same signal is applied to it several different times. [Ref. 6] One technique used to get around this is to ignore a set number of the least significant bits generated. The assumption is that the result will differ in only a known number of the least significant bits. An alternative approach is the mid-value select technique. The voter basically just selects the middle value of the three inputs as shown in Figure 8. Essentially, it is the same concept as a majority voter but is necessary when the three values may have slight perturbations between them. The middle value is chosen

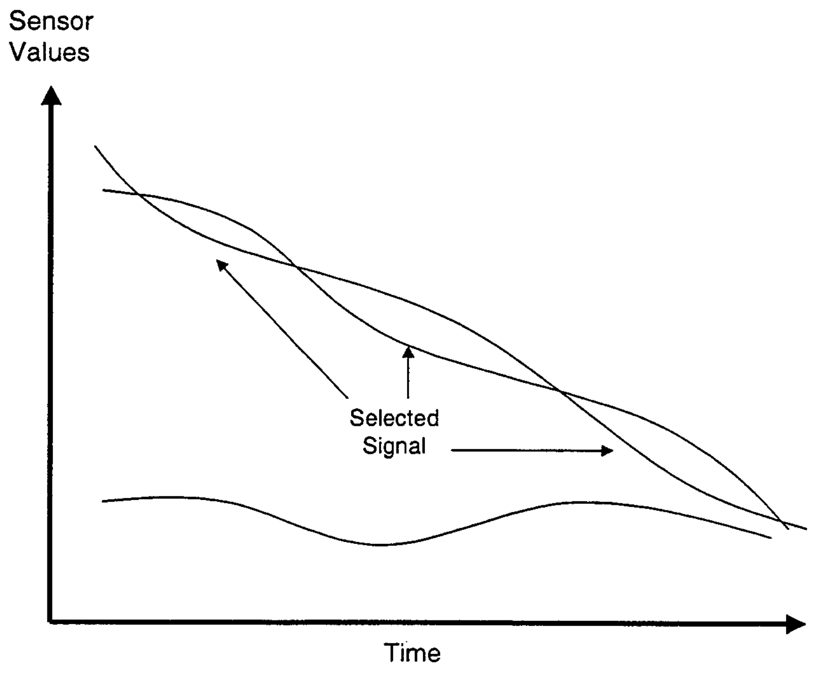


Figure 8. Mid-value select technique. After Ref. [6].

because an assumption is made that only one of the inputs can be faulty at one time. Thus, since minor perturbations are expected the middle value will always be one from a "good" input. The middle value is chosen instead of taking an average of the three inputs. This is because in the event that one input is clearly faulty as shown in Figure 8, the average would be adversely affected. In effect, the faulty input is ignored by selecting the middle value.

Another problem that must be realized in a TMR system with majority voting is that identical errors in two of the modules will have to be tolerated. The errors will produce results that when passed to the voter will be selected as the majority. The possibility of this occurring and the consequences would definitely have to be investigated depending on the application.

A significant danger of incorporating redundancy into a system is that the overall system reliability could be reduced, due to the increased number of components. If the redundant systems are not themselves reliable, there is little hope of improving the reliability of the system. [Ref. 7] For example, Wakerly notes that constructing a voting component for three microprocessors in a TMR structure could conceivably require 14 integrated circuit packages constructed from the same (unreliable) technology as the three microprocessor packages, and hence would lead to a system with lower reliability than that of a single microprocessor chip. [Ref. 8] In addition, on a PC board, solder connections can be one of the largest sources of failure.

On the other hand, given that the redundant components are sufficiently reliable and the additional logic required is at least as reliable as the redundant modules, TMR provides a viable technique for improving overall system reliability in critical applications. [Ref. 4]

B. TRIPLE MODULAR REDUNDANT MICROPROCESSOR DESIGN

Having reviewed the concepts of TMR, what follows is a description of how they might be employed with three microprocessors. Also, having chosen to build the Testbed using the IDT R3081 RISC Microprocessor discussed in Chapters I and II, it is useful to examine what is necessary in constructing a board with three R3081's operating in a TMR design.

Figure 9 shows a block diagram of a simple system using a single R3081 processor. The multiplexed address/data bus of the R3081 is demultiplexed through the use of address latches and data buffers/transceivers. The address bus and the control bus are then used by the memory controller to access the memory blocks. A typical design similar to Figure 9 is described in detail in Ref. [9].

Expounding on this simple system, Figure 10 shows a block diagram of a TMR system using three R3081 processors. Figure 10 shows the additional hardware blocks necessary to implement majority voting of the address, data, and control

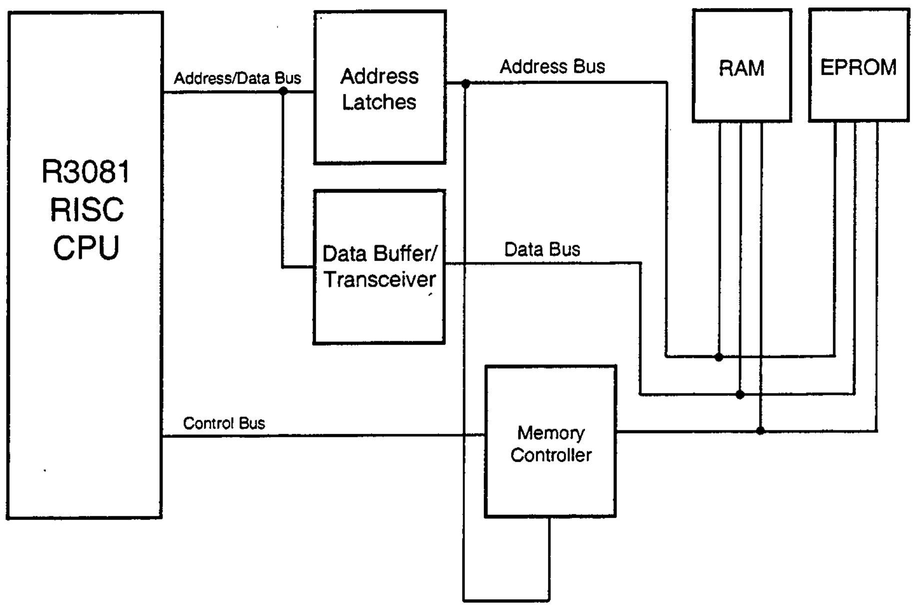


Figure 9. Simple R3081 Board Design. After Ref. [9].

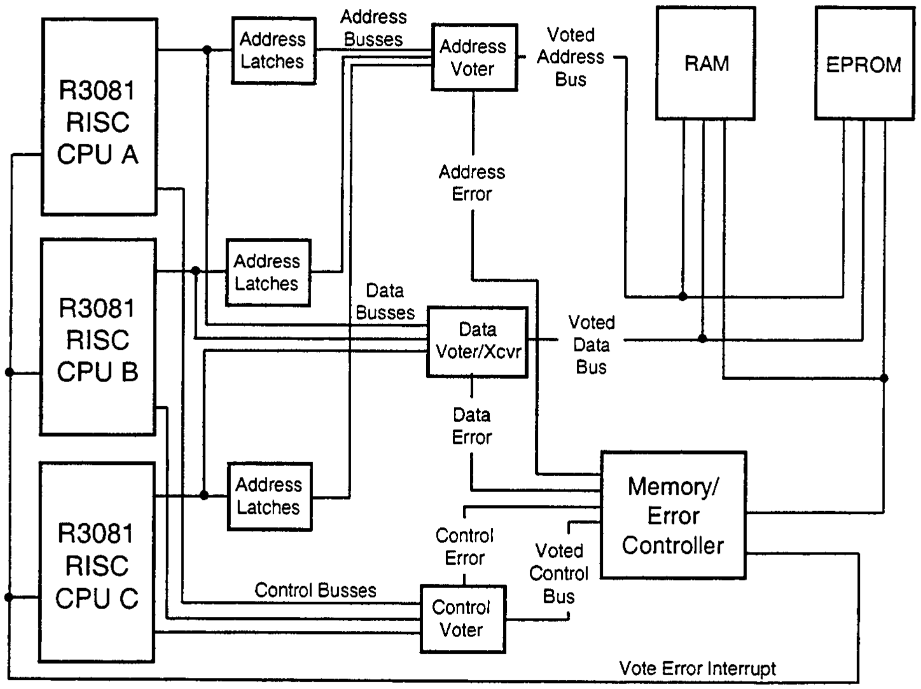


Figure 10. TMR R3081 Board Design.

busses and how the voted busses are then used in the remainder of the system.

A significant issue when using three microprocessors in a TMR design is the synchronization of the processors, briefly described in the preceding section, Voting Issues (Section A, Subsection 2, of this chapter). The IDT R3081 contains an output from the processor which is the System Reference Clock, SysClk*. This clock is used to control state transitions in the read buffer, write buffer, memory controller, and bus interface unit internal to the processor. As such it is used as timing reference by the external memory system. The frequency of this clock can be either the same as the CPU cycle rate, or one-half that frequency. The frequency of this clock is selectable during the processor reset initialization. [Ref. 5]

The R3081 does not have a guaranteed relationship between the input clock and the SysClk* System Reference Clock. However, it is possible to ensure the phase of this output reference clock allowing the multiple processors to be in the same phase. The IDT R3081 contains internal logic as part of its reset state machine, which forces the System Reference Clock, SysClk*, into a known state. [Ref. 5] Thus in a system using multiple R3081 processors with their System Reference Clocks operating at the same frequency as the CPU cycle rate, the negation of the Reset* input to the processors is sufficient to ensure that the System Reference Clocks from each processor are operating in the same phase. This assumes that the three processors are driven by the same input clock. [Ref. 5] If the Output Reference Clocks are operating at one-half of the frequency of the CPU cycle rate, additional steps are necessary to ensure synchronization between the System Reference Clocks from multiple CPUs.

In order to take full advantage of the TMR design to allow error analysis, FIFOs dedicated to each processor were incorporated as shown in Figure 11. The FIFOs allow the capturing of the address, control, and data bus information from each processor before it is passed to the majority voters as shown in Figure 10.

Detailed descriptions of the blocks shown in Figures 10 and 11 and how they are implemented in the Testbed design are discussed in the next chapter.

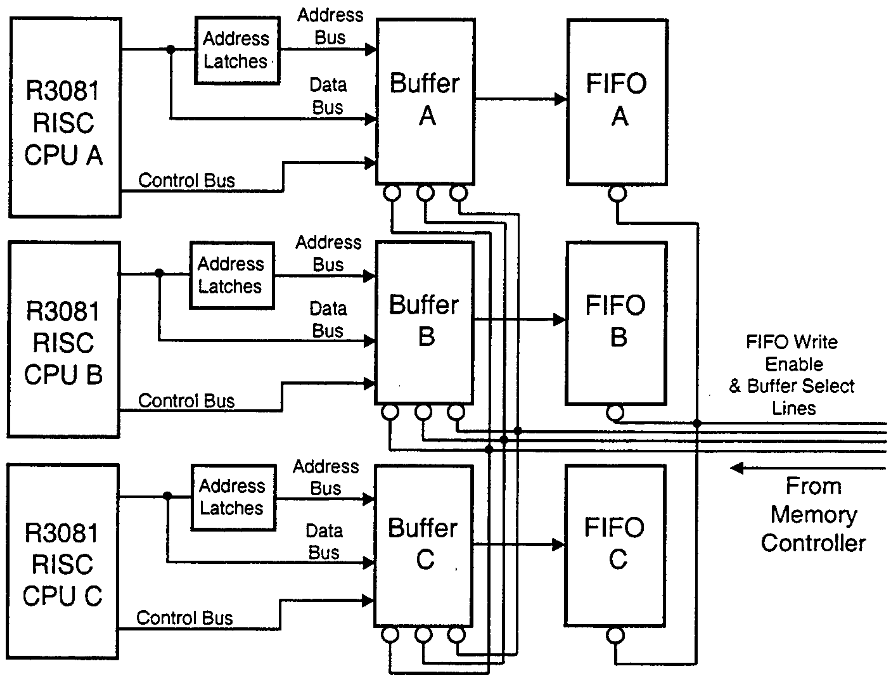


Figure 11. Testbed FIFO Interface.

IV. TMR TESTBED DESIGN

A. OVERVIEW

In order to observe the performance and behavior of a microprocessor in the presence of radiation induced single event upsets (SEUs), the address, data, and control busses must be monitored. This is because in a general purpose microprocessor there is not an efficient built-in mechanism to indicate to external devices and/or observers that an SEU induced error has occurred. This is particularly true in the case where one or more bits in a word of data are flipped. SEU induced errors may cause the processor to "lock up" or "crash," which is detectable, but is of little use when trying to trouble-shoot and/or monitor the performance of the system.

Monitoring of the address and data busses presents another problem. Without a separate entity which is deemed, or assumed, to be error free there is not a way to tell if the information that appears on the busses is error free or not. In addition, in the presence of radiation induced SEUs, the ability to correct such faults once detected is a desirable characteristic.

In this testbed design, triple modular redundancy (TMR) was chosen to allow the monitoring of three identical microprocessors running identical programs. The majority voting used in conjunction with TMR allows detection of an SEU which has been manifested as a disagreement between the address, data, and control busses of the three processors. The majority voter also allows the masking of these SEU induced disagreements. The address, data, and control bus information from the two microprocessors which are in agreement is used to start, control, and complete each bus cycle.

This assumes that identical faults, or errors, will not occur in two different microprocessors and produce the same erroneous results on their associated busses. If this occurred, then the majority would be in an error state. The same argument applies for identical faults in all three processors. The following sections describe the Testbed TMR functionality and the use of dedicated FIFOs for error analysis.

1. Testbed Operation Summary

The testbed contains three IDT R3081 RISC microprocessors executing the same program and interrupt service routines. Each processor has a dedicated FIFO memory to capture the address, control, and data bus information during each bus cycle. The address, data, and control busses from the three processors are then combined into single address, control, and data busses via majority voters. These voted busses are then used by a single memory/error cycle controller to access the same ROM and RAM.

a. Normal (Error Free) Operation

At the beginning of a bus cycle (Read, Burst Read, or Write), the address is latched from each processor's A/D bus. Voting commences on the address busses while they are simultaneously written to each FIFO.

Control lines are next sampled from each processor. Voting commences on the control busses while they are simultaneously written to each FIFO.

Data on the A/D bus from each processor is voted (during a Write cycle only). Data on the A/D busses from each processor during both Read and Write bus cycles, including Burst Read, are written to each FIFO.

If no error is detected (address, control, or data), then the current bus cycle finishes normally.

b. Error Detection

Errors are detected by majority voting of the address, control, and data busses from each processor. If an error is detected, the current bus cycle is allowed to complete before generating an interrupt. The error is masked during Read and Write operations through the majority voter. However, the address, control, and data bus information associated with each processor before voting occurs will have been placed in each FIFO for analysis. Upon completion of the current cycle, an interrupt is generated and synchronously supplied to each processor.

c. Error Correction

Upon receipt of an interrupt, each processor executes the same interrupt service routine. The beginning of this routine is signaled by initiating a write to "dummy" address $1F80xxxx_H$. The dummy address is recognized by the address decoder and a dedicated chip select is asserted. This chip select is in turn recognized by the memory/error cycle controller. The memory/error cycle controller clears the current interrupt and disables subsequent vote error interrupts while the interrupt routine executes.

The internal general purpose registers, configuration registers, and instruction and data caches are written to a reserved location in RAM. While this occurs, all internal information associated with each processor is written to a dedicated FIFO. The majority voter masks the error in the faulty processor and the "corrected" information, based on the majority of the two agreeing processors, is written to RAM. All internal registers and caches in each processor are then filled by reading the reserved locations in RAM. The "faulty" processor will now have been "corrected" and re-synchronized with the other two processors.

The processors signal the end of the interrupt service routine by initiating another write to "dummy" address $1F80xxxx_H$. The memory/error cycle controller will then re-enable vote error interrupts, and the next bus cycle begins.

d. Error Monitoring

The operation of the Testbed is monitored via an outside interface system. This outside system reads the contents of the FIFOs associated with each processor. Address, control, and data bus information from each processor are placed in FIFOs during non-error bus cycles.

Upon detection of an error and interrupt handler execution, all internal registers and caches for each processor are written to the dedicated FIFOs.

The FIFOs now contain the information necessary to detect which processor was in error and what the processors were doing at the time the error occurred.

2. IDT R3081 Simulation

We do not have a model of the complete R3081 RISC Microprocessor for simulation of the Testbed design. Therefore, in order to develop the concept of this design we modeled the behavior of the IDT R3081 multiplexed address/data bus and associated control lines using the Verilog Hardware Description Language [Ref. 10]. The remaining sections of this chapter describe in detail each of the blocks in the Testbed design.

In the descriptions of the blocks and in the associated figures, the following convention has been used. Signal and bus names which are bold and italicized, ***FORCE_A*** for example, are intended to match the same signal and bus names in the overall schematic in Appendix A for ease in cross referencing. In addition, signal and bus names which begin with an underscore, ***_ALE*** for example, represent signals which come from each of the three processors. Thus ***_ALE*** represents ***A_ALE***, ***B_ALE***, and ***C_ALE***, for example.

B. IDT R3081 BUS INTERFACE

In this section, we will demonstrate that the bus interface simulation matches the manufacturers design specifications for the R3081.

The datasheet for the IDT R3081 RISC Microprocessor [Ref. 11] was used in conjunction with the R3081 Hardware Users Manual [Ref. 5]. The single datum (word or byte) Read, Burst Read, and Write bus cycle timing diagrams and timing parameters were analyzed and used to simulate the

R3081 bus interface. Figures 12, 13, and 14 are the bus cycles obtained from these references.

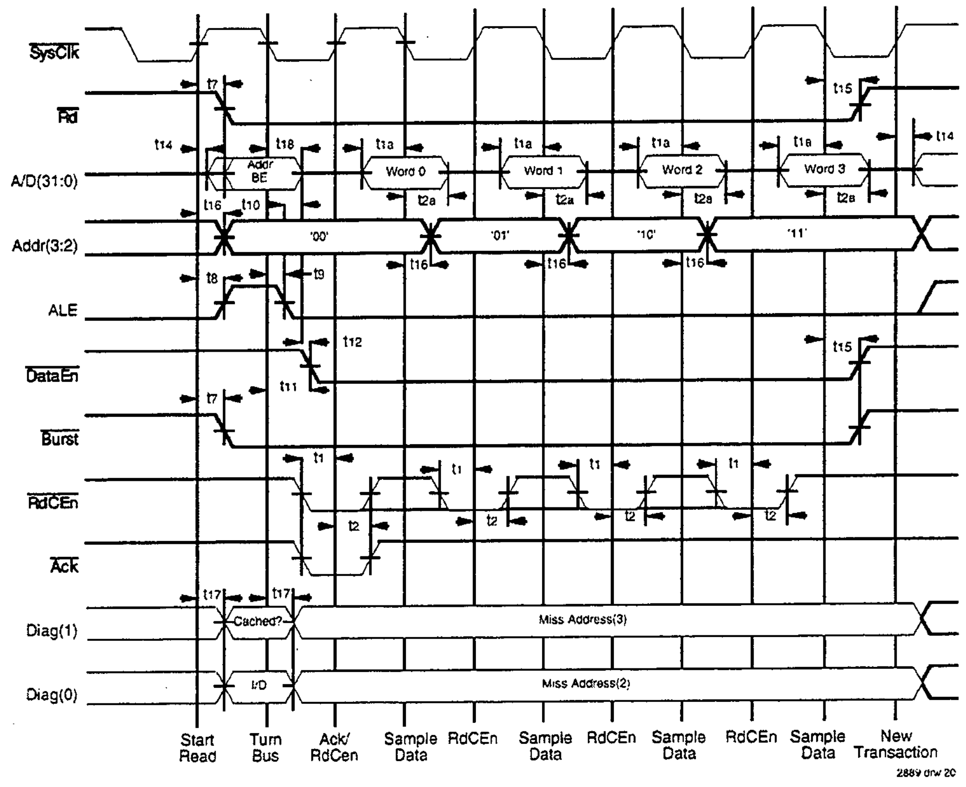


Figure 12. IDT R0381 Burst Read Cycle. From Ref. [9].

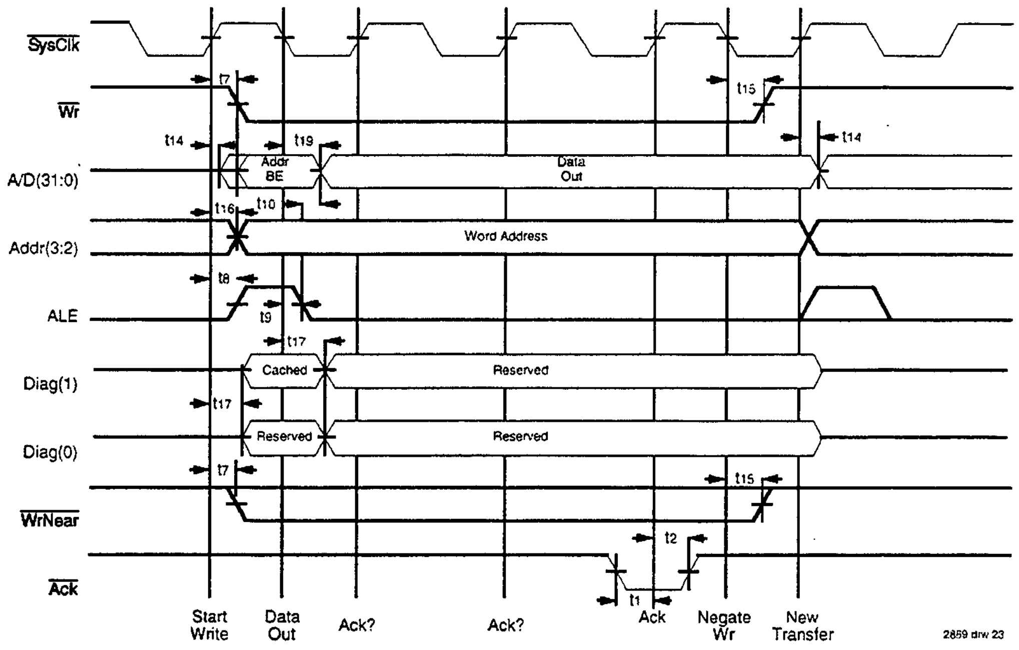


Figure 13. IDT R3081 Write Cycle. From Ref. [9].

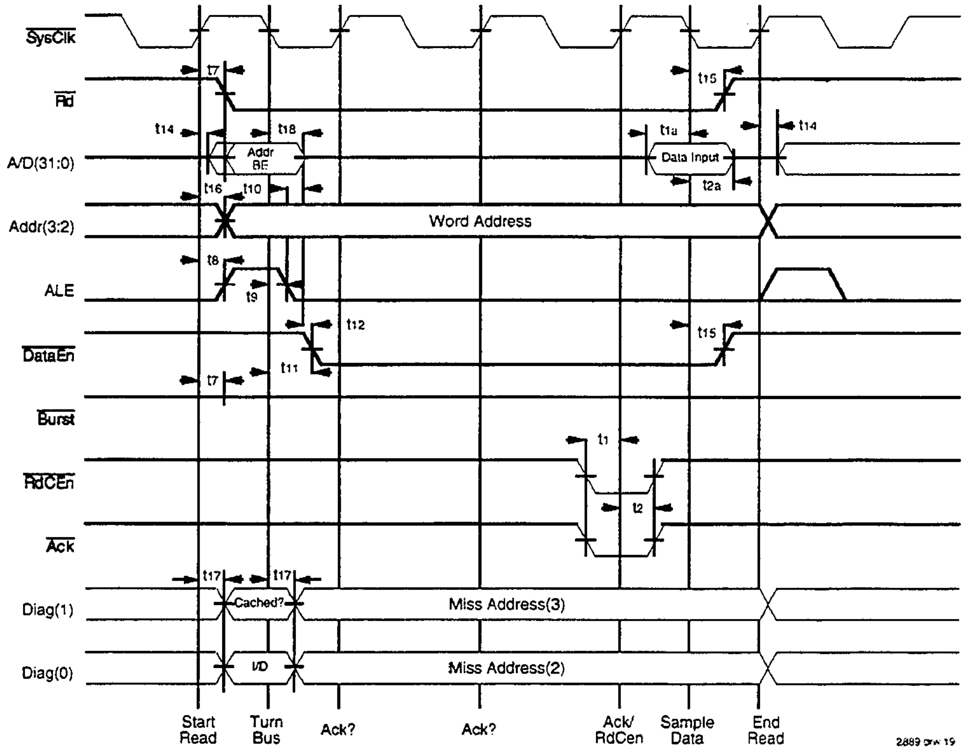


Figure 14. IDT R3081 Single Datum Read. From Ref. [9].

The Diag(1) and Diag(0) signals shown in Figures 12, 13, and 14 were not modeled. These two pins are useful in the initial debug of R30xx family based systems. [Ref. 5] Although they are not control lines, in an actual implementation of the Testbed, these lines could easily be added as part of the control bus from each microprocessor and passed to the control majority voter. They are not needed to control the bus/memory interface. However, they could be used as additional status lines to detect differences among the three processors.

Figure 15 shows the R3081 bus interface simulator built in Cadence Concept™ Schematics and the Verilog Hardware Description Language. The associated Verilog code is contained in Appendix C, Section A. The three pins on the

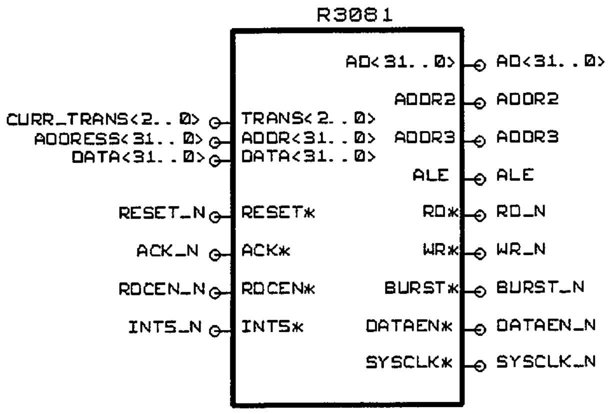


Figure 15. IDT R3081 Bus Interface Simulator.

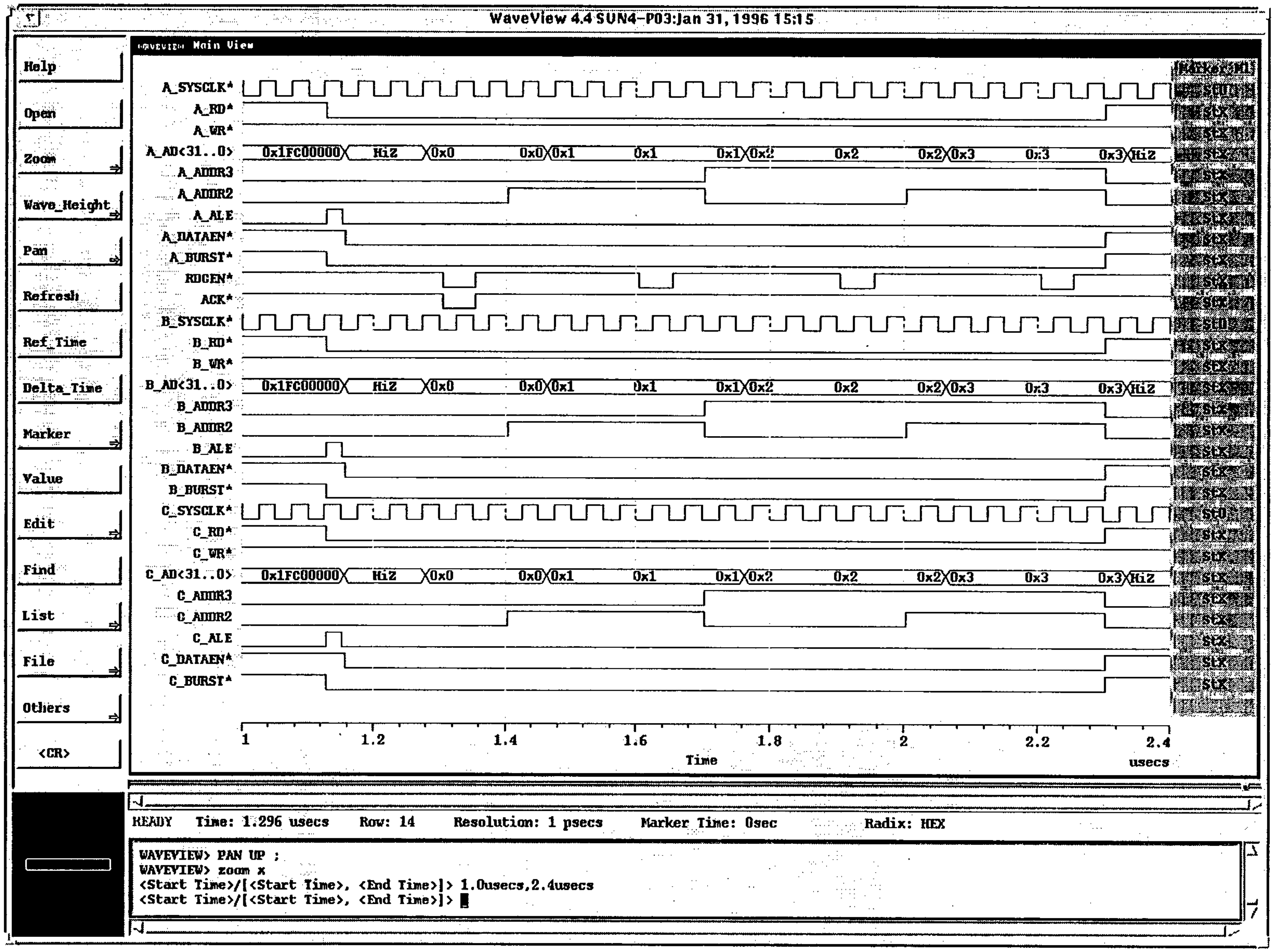
simulator labeled ***TRANS<2..0>, ADDR<31..0>,*** and ***DATA<31..0>*** are not pins on an actual R3081 device. These pins are used during simulations to force the simulator to execute a specified bus cycle. ***TRANS<2..0>*** is used to specify either Byte Read, Word Read, Burst Read, Byte Write, or Word Write bus cycles. ***ADDR<31..0>*** is used to specify the address of the current bus cycle. If the current bus cycle specified is a Burst Read, then ***ADDR<31..0>*** specifies the initial word address. ***DATA<31..0>*** is used to specify the data to be used during Write bus cycles. By using three separate simulators and specifying each of the above three signals separately to each simulator, faults can be injected into the system.

Figures 16, 17, and 18 show the simulated address/data bus and control line behavior. Extra wait states; i.e., additional system reference clock cycles, have been added to each bus cycle. The extra wait states allow FIFO memories dedicated to each microprocessor to grab the address, control, and data bus information. In addition, in these three figures the address/data bus and control lines from each of the three microprocessors are displayed to show they are synchronized with one another.

In Figure 16, the Burst Read cycle is initiated at the falling edge of the ***_RD**** and ***_BURST**** lines from each microprocessor. In this particular example, the address

$1FC00000_H$ is placed on the multiplexed address/data bus, ***_AD<31..0>,*** by each processor. After this address is latched using the ***_ALE*** signals from each processor, the first word of data appears on the ***_AD<31..0>*** bus after a short delay from the memory. The four contiguous words of memory read during this bus cycle are obtained by providing the initial address, $1FC00000_H$ in this case, and strobing the ***_ADDR3*** and ***_ADDR2*** lines so that they count in binary 00, 01, 10, and 11. In addition, the memory controller strobes the ***RDCEN**** line, which is supplied to all three microprocessors, four times indicating when the expected word from memory has been placed on the bus. The burst read cycle is completed at the rising edge of the ***_RD**** and ***_BURST**** signals. In the example in Figure 16 the four addresses read are $1FC00000_H$, $1FC00004_H$, $1FC00008_H$, and $1FC0000C_H$. In this design, the addresses $1FC00000_H$ through $1FC0xxxx_H$ are decoded to be read only memory (ROM). The four words read contained the data 00000000_H, 00000001_H, 00000002_H, and 00000003_H, respectively. This correctly corresponds to the data which has been programmed into the EPROM. See Appendix C, Section I.

In Figure 17, the Write cycle is initiated at the falling edge of the ***_WR**** lines from each microprocessor.



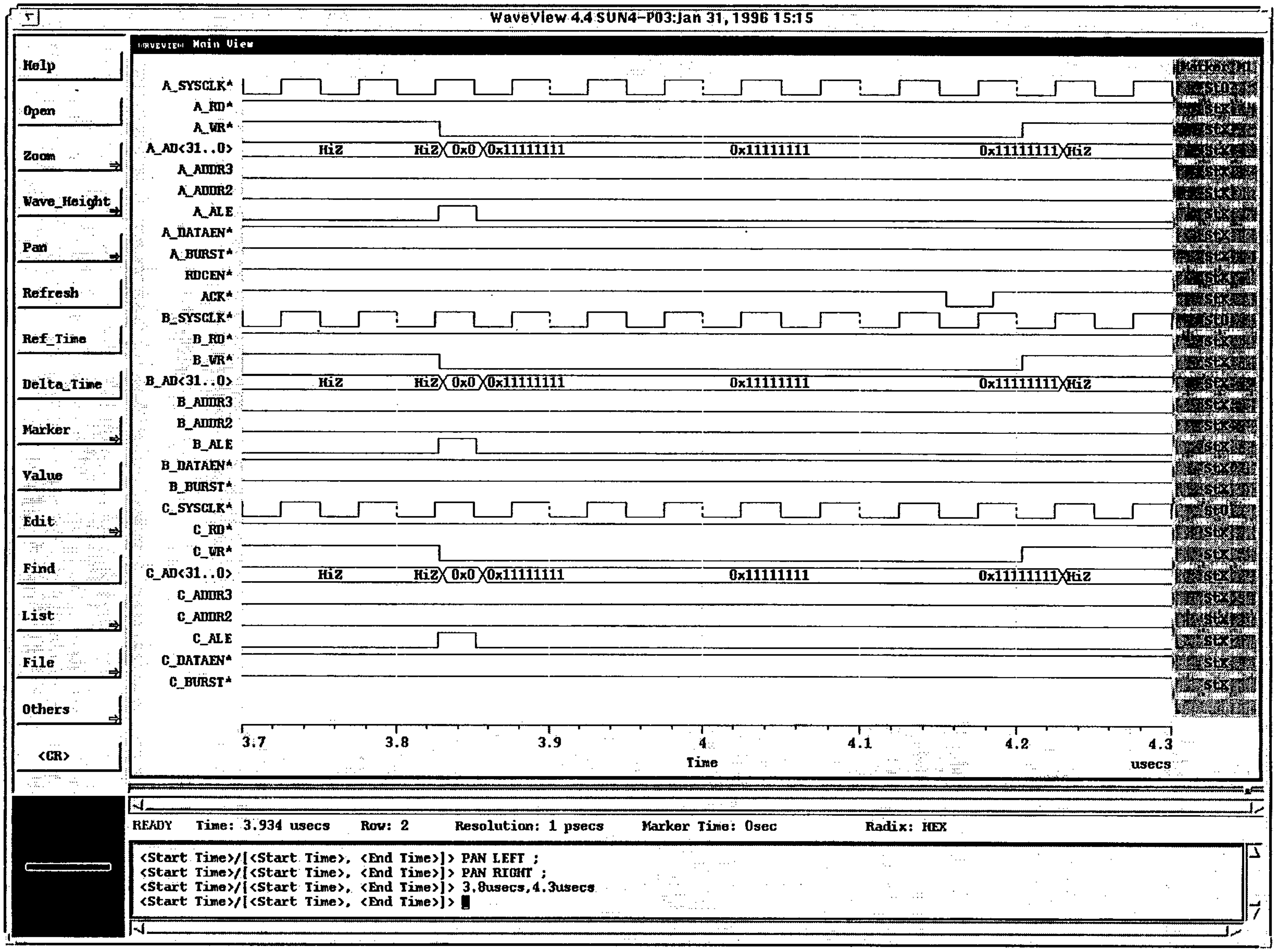
```
READY   Time: 1.296 usecs   Row: 14   Resolution: 1 psecs   Marker Time: 0sec   Radix: HEX
```

```
WAVEVIEW> PAN UP ;
WAVEVIEW> zoom x
<Start Time>/[<Start Time>, <End Time>]> 1.0usecs,2.4usecs
<Start Time>/[<Start Time>, <End Time>]>
```

Figure 16. Simulated R3081 Burst Read Cycle.

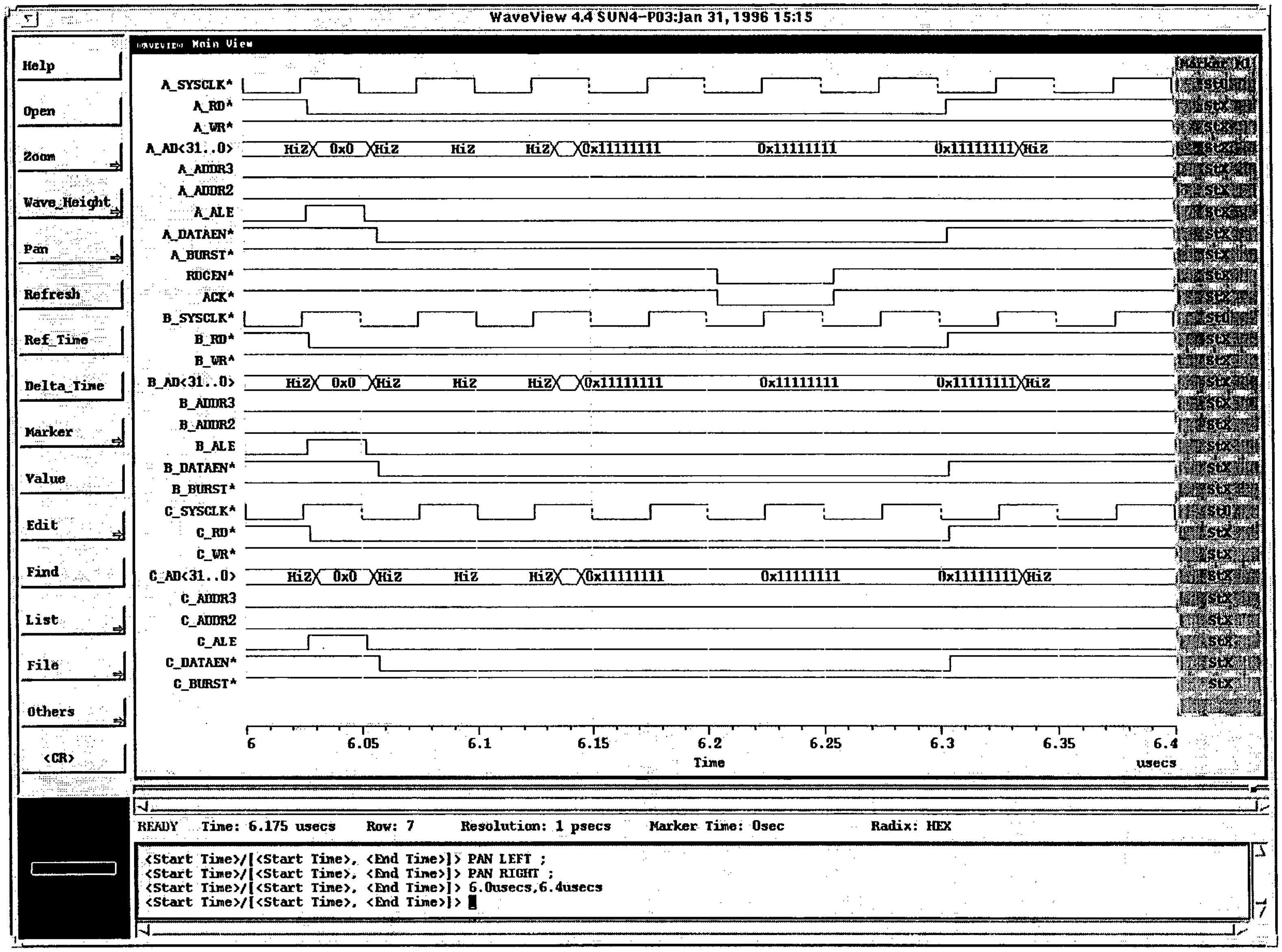
In this particular example, the address 00000000_H is placed on the multiplexed address/data bus, ***_AD<31..0>,*** by each processor. After this address is latched using the ***_ALE*** signals from each processor, the data to be written appears on the bus. In this example, the data to be written is 11111111_H. The ***ACK**** signal, which is returned from the memory controller, indicates the write has been completed. The write cycle is completed at the rising edge of the ***_WR**** signal. In the TMR Testbed design, addresses 00000000_H through $0007FFFF_H$ correspond to random access memory (RAM). Therefore, in this example, 11111111_H has been written to RAM at address 00000000_H.

In Figure 18, the single datum Word Read cycle is initiated at the falling edge of the ***_RD**** lines from each microprocessor. In this particular example, the address 00000000_H is placed on the multiplexed address/data bus, ***_AD<31..0>,*** by each processor. After this address is latched using the ***_ALE*** signals from each processor, the data appears on the bus after some delay. The ***RDCEN**** line from the memory controller indicates that the address/data bus contains valid data. The read cycle is completed at the rising edge of the ***_RD**** lines. In this example, 11111111_H has been read from RAM at address 00000000_H. This correctly corresponds with the 11111111_H written to address 00000000_H



```
READY   Time: 3.934 usecs   Row: 2   Resolution: 1 psecs   Marker Time: 0sec   Radix: HEX
<Start Time>/[<Start Time>, <End Time>]> PAN LEFT ;
<Start Time>/[<Start Time>, <End Time>]> PAN RIGHT ;
<Start Time>/[<Start Time>, <End Time>]> 3.8usecs,4.3usecs
<Start Time>/[<Start Time>, <End Time>]> 
```

Figure 17. Simulated R3081 Write Cycle.



```
READY   Time: 6.175 usecs    Row: 7     Resolution: 1 psecs     Marker Time: 0sec        Radix: HEX

<Start Time>/[<Start Time>, <End Time>]> PAN LEFT ;
<Start Time>/[<Start Time>, <End Time>]> PAN RIGHT ;
<Start Time>/[<Start Time>, <End Time>]> 6.0usecs,6.4usecs
<Start Time>/[<Start Time>, <End Time>]> █
```

Figure 18. Simulated R3081 Read Cycle.

in the previous example Write cycle description and in Figure 17.

C. ADDRESS/DATA BUS DEMULTIPLEXING

The multiplexed 32-bit address/data bus of each of the three microprocessors is demultiplexed using the address latch enable, ***_ALE***, signal [Ref. 5] from each processor. The schematic diagram of the demultiplexer is contained in Appendix B, Section A. Figure 19 is a block diagram of the demultiplexer.

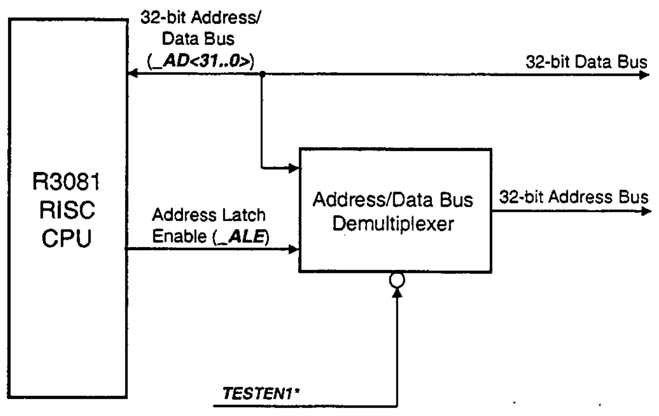


Figure 19. Address/Data Bus Demultiplexing.

Each 32-bit demultiplexer makes use of four 8-bit FCT373 transparent latches. [Ref. 9] During each bus cycle (Read, Burst Read, or Write) the address is placed on the ***_AD<31..0>*** bus of each processor at the beginning of the cycle. While the ***_ALE*** signals are HIGH, the transparent latches allow the address information to pass to the 32-bit address voter. This allows the address information to be voted and passed to the memory/address decoder as soon as it becomes available. When the ***_ALE*** signals transition from HIGH to LOW, the address information is latched to the associated 32-bit address bus. Subsequent changes on the ***_AD<31..0>*** busses do not affect the state of the address busses until the next ***_ALE*** transition from LOW to HIGH, which occurs during the next bus cycle. The ***TESTEN1**** line, which is supplied to each demultiplexer, can be used to place the address bus, or output of each demultiplexer, in a high impedance state for testing. During normal operations, the ***TESTEN1**** line should be held LOW. The schematic diagram of the three microprocessors, the demultiplexers, and the associated connections is contained in Appendix A.

D. DATA BUS VOTING

The ***_AD<31..0>*** bus from each microprocessor is considered to be the data bus after the transition of the ALE signal from HIGH to LOW during each bus cycle. The 32-bit data busses from each processor are passed to a 32-bit majority voter/transceiver. Figure 20 is a block diagram of the data bus voter/transceiver.

During a Write cycle, the three 32-bit data busses are voted to produce a single 32-bit data bus. However, during a Read, or Burst Read, bus cycle the data read from memory

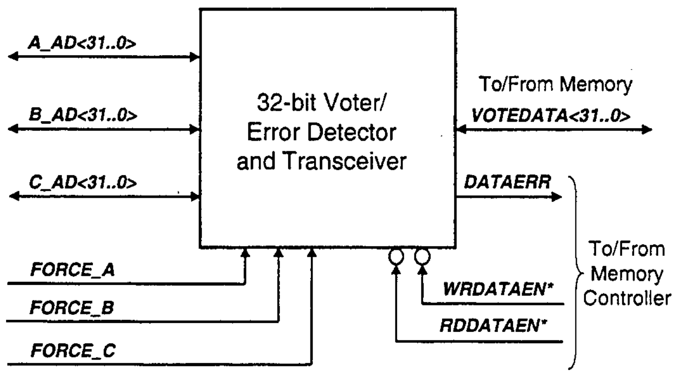


Figure 20. Data Bus Voting.

must be allowed to pass back to the three ***_AD<31..0>*** busses and on to the three microprocessors. This is accomplished via the ***RDDATAEN**** and ***WRDATAEN**** control lines from the memory enable controller. While the ***WRDATAEN**** signal is LOW, the three data busses are voted and passed to the single data bus. While the ***RDDATAEN**** line is LOW, the data on the single bus which has been read from memory is allowed to pass back through to the three microprocessors. Voting of the data busses occurs only during a Write cycle and when ***WRDATAEN**** is LOW. The ***WRDATEN**** and ***RDDATAEN**** signals are mutually exclusive (when one is HIGH, the other is LOW). If an error is detected on one of the data busses supplied to the voter, the signal ***DATAERR*** goes HIGH.

In addition, the majority voter/transceiver uses three input lines (***FORCE_A***, ***FORCE_B***, and ***FORCE_C***) which, when pulled HIGH, force the data from the respective bus through to the output data bus. When one of these signals is pulled HIGH, voting errors are not detected or signaled. These signals should all be held LOW during normal operations.

The schematic for the 32-bit majority voter/transceiver and associated Verilog code are contained in Appendix C, Section B.

E. ADDRESS BUS VOTING

The output of the three demultiplexers is considered to be the address bus associated with each processor. Once a bus cycle has initiated and the ***_ALE*** has transitioned from HIGH to LOW, the address bus holds the address information until the LOW to HIGH transition of ***_ALE*** during the next bus cycle. The address bus from each demultiplexer is passed to a 32-bit majority voter. This majority voter operates similarly to that of the majority voter/transceiver described in the previous section except there is no associated transceiver operation or control lines. Figure 21 is a block diagram of the address voter. If an error is detected on one of the address busses supplied to the voter, the signal ***ADDRERR*** goes HIGH.

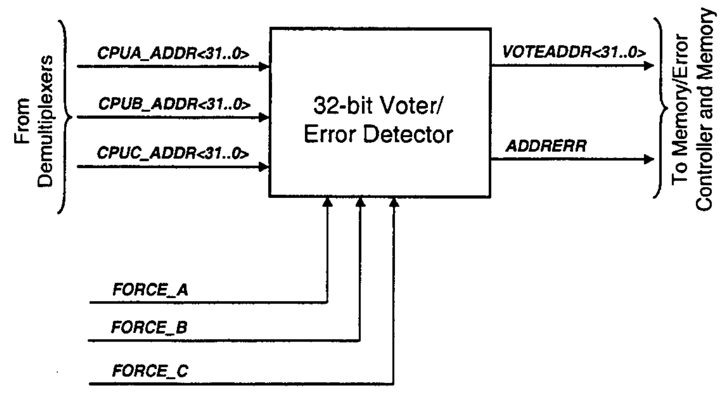


Figure 21. Address Bus Voting.

The schematic for the 32-bit majority voter and associated Verilog code are contained in Appendix C, Section D.

F. CONTROL BUS VOTING

Six control lines from each of the three processors are voted using an 8-bit majority voter. The six control lines voted are ***_ADDR2, _ADDR3, _RD*, _WR*, _BURST**,** and ***_DATAEN****. The other two inputs to the 8-bit voter are not used and are held LOW. These control lines are voted to produce a single control bus. Figure 22 is a block diagram of the control bus voter. This majority voter operates similarly to that of the majority voter/transceiver described in Section D except there is no associated transceiver operation or control lines. If an error is detected on one of the control lines supplied to the voter, the signal ***CONTERR*** goes HIGH.

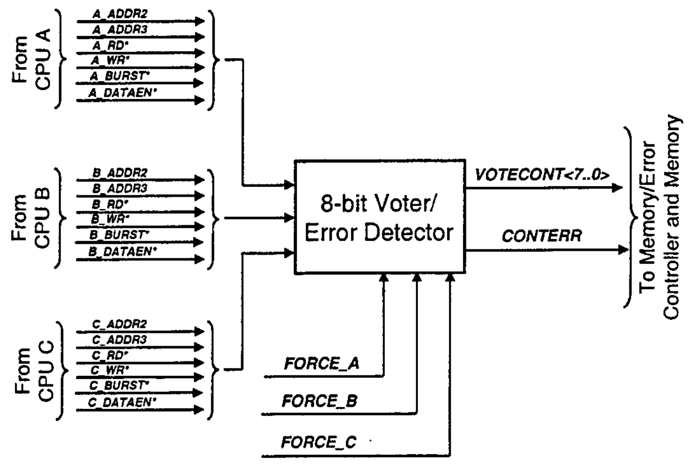


Figure 22. Control Bus Voting.

The schematic for the 8-bit majority voter and associated Verilog code are contained in Appendix C, Section C.

G. ADDRESS DECODER

The address decoder uses the voted address bus, ***VOTEADDR<31..17>***, to generate chip selects. The address decoder does not wait for ***_ALE*** to begin generating the chip selects. This is done to achieve better performance since the chip select outputs will be generated earlier in the bus cycle. As a side effect, however, the chip select outputs may tend to "glitch" as a valid address is driven. Thus, the Read Enables and Write Enables seen in the memory system must be synchronized so they are valid only when the CPUs are attempting a read or write transfer. This combination allows maximum performance because address and chip selects are seen early in the bus cycle but the Read and Write signals are synchronized to ensure proper system operation. [Ref. 9] Figure 23 is a block diagram of the address decoder.

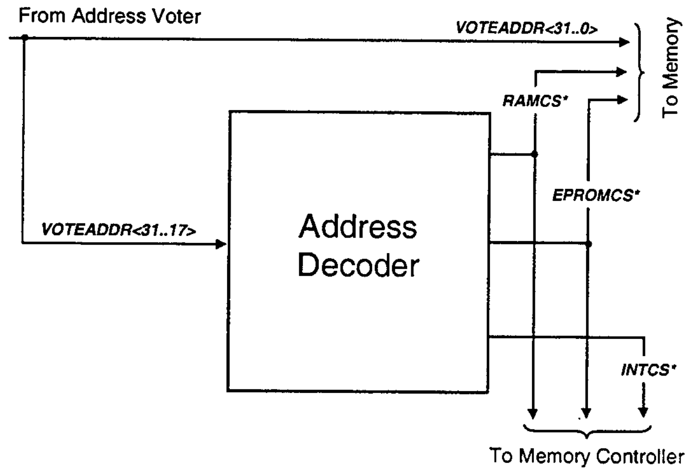


Figure 23. Address Decoder.

The schematic for the memory/address decoder and associated Verilog code are contained in Appendix C, Section E.

H. MEMORY/ERROR CYCLE CONTROLLER

The memory cycle controller provides a wait-state generator which stalls the bus interfaces of the three processors so that various types and speeds of memories can be used. [Ref. 9] This also allows the additional wait-states required for the FIFO interface described later. Figure 24 is a block diagram of the memory/error cycle controller. The memory/error cycle controller is composed

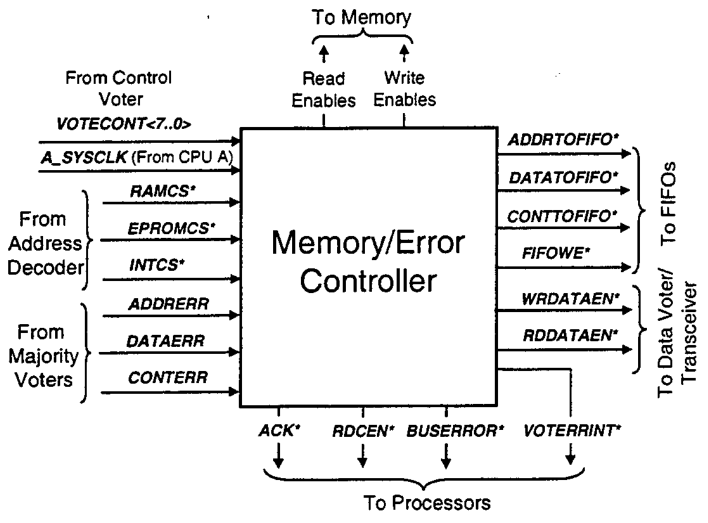


Figure 24. Memory/Error Cycle Controller.

of three subsections. The basic RAM/ROM subsection generates the appropriate timing signals such as ***ACK*,*** ***RDCEN*,*** and ***BUSERROR**** for operating the R3081 bus interface as well as the necessary write and read enables for accessing the RAM/ROM. The FIFO memory cycle controller generates the signals necessary for capturing the state of each processor in its dedicated FIFO at the appropriate times during each cycle. The error cycle controller monitors the vote error signals from the address, data, and control bus majority voters. If an error is detected, it generates an interrupt to the processors. It also disables the vote error interrupts while the interrupt handler routine is executed by the processors. The schematics for the memory/error cycle and memory enable controllers and associated Verilog code are contained in Appendix C, Sections F and G.

1. RAM/ROM Cycle Controller

The basic state machine looks for the start of a read or write bus cycle by looking for a negative edge of ***VOTRD**** or ***VOTWR**** from the control bus majority voter. When a bus cycle is initiated, the state machine starts a 5-bit up counter, counter<4..0>. The counter then increments on each ***SYSCLK**** rising edge. This counter is then used as the timing master for all other control signals generated by the state machine. [Ref. 9]

A synchronous decoder, ***CYCEND****, is used to tell the counter when the end of a memory cycle occurs. ***CYCEND**** is used to synchronously reset the state machine when a positive edge of ***VOTRD**** or ***VOTWR**** is expected. Another output, ***ENSTART****, is used to start the byte enables generated by the memory enable controller. [Ref. 9]

Other outputs from the memory cycle controller include cycle termination inputs ***RDCEN****, ***ACK****, and ***BUSERROR****. On a read transfer, ***VOTBURST**** from the control bus voter and the current active chip select from the address decoder are used to determine the timing and quantity of ***RDCEN**** signals to be asserted. ***ACK**** is asserted at the end of a write cycle to indicate completion of the transfer. ***BUSERROR**** is used to end an undecoded memory cycle. [Ref. 9]

2. FIFO Memory Cycle Controller

In order to provide the ability to observe the status of each processor before, during, and after an error cycle, the address, control, and data busses (before the majority voters) from each processor are written to a dedicated FIFO memory. The state machine in the memory cycle controller is used to generate the outputs ***ADDRTOFIFO****, ***CONTTOFIFO****, ***DATATOFIFO****, and ***FIFOWE****. Figure 25 shows a block diagram of the FIFO dedicated to processor A. A similar arrangement is used for the FIFOs dedicated to processors B and C. The use of the memory cycle state machine ensures the timing of these signals are synchronized with the current bus cycle and that during a Burst Read bus operation, the address, control, and data busses are written to the FIFOs four times.

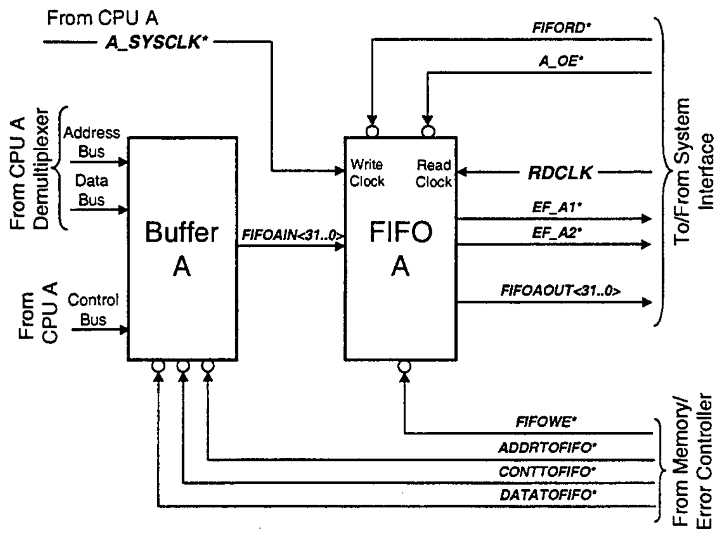


Figure 25. FIFO Controls.

The ***ADDRTOFIFO*, CONTTOFIFO****, and ***DATATOFIFO**** outputs synchronously select when to provide the address bus, control bus, and data bus respectively to the FIFO associated with each processor. Since the address is the first bus to stabilize, ***ADDRTOFIFO**** is asserted first. This is followed by ***CONTTOFIFO**** and then ***DATATOFIFO****. ***FIFOWE**** is the actual write enable supplied to the three FIFOs.

When ***ADDRTOFIFO**** is asserted, the address bus from each processor is supplied to its associated FIFO and written at the rising edge of ***FIFOWE****. This is followed by ***CONTTOFIFO**** and ***DATATOFIFO****, in turn.

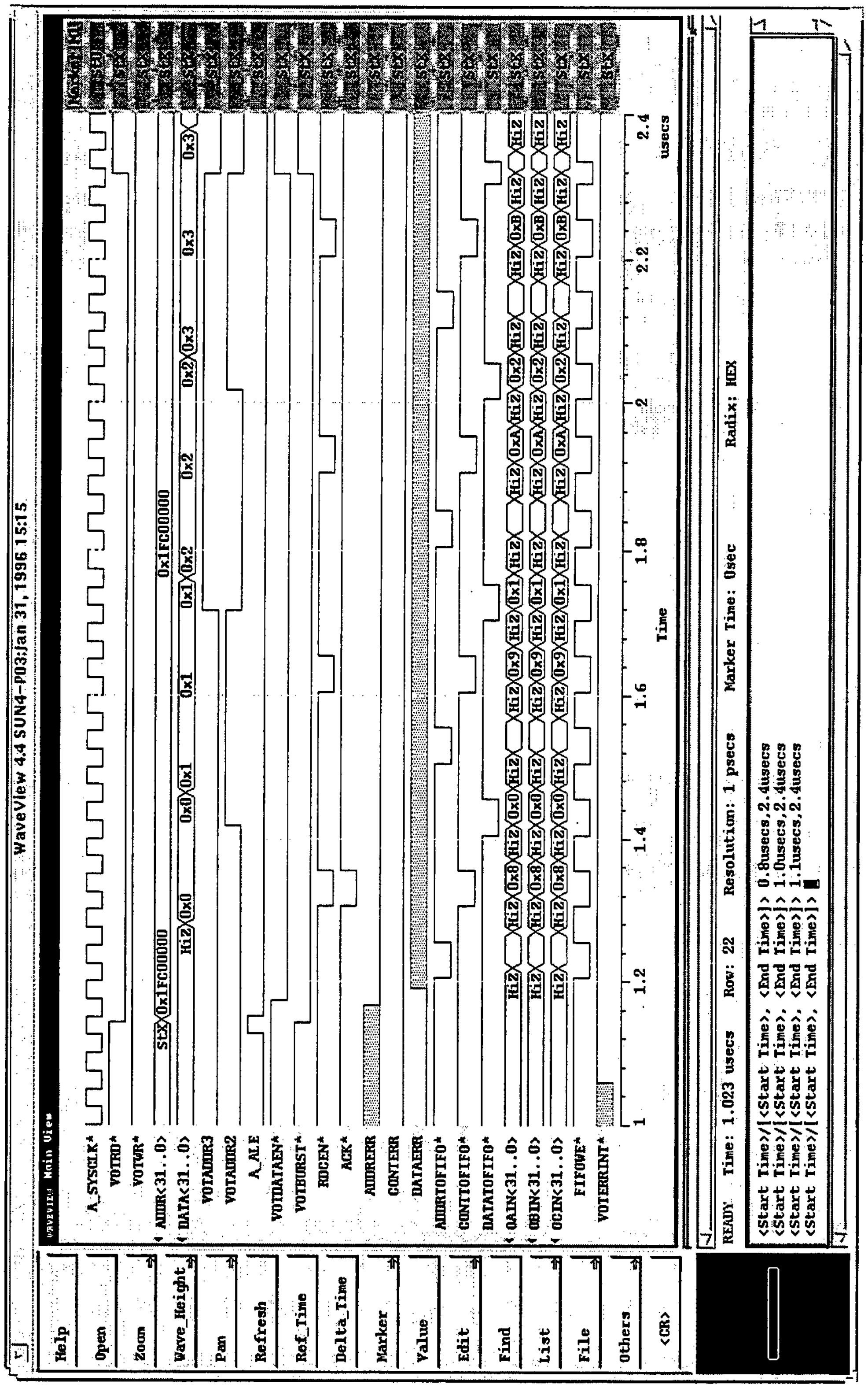
Figures 26, 27, and 28 show the operation of these FIFO controls during a Burst Read, Write, and single word Read respectively.

3. Error Cycle Controller

The memory cycle controller state machine also controls the generation of an interrupt which is supplied to each processor at the detection of a vote error (***ADDRERR, CONTERR,*** or ***DATAERR***).

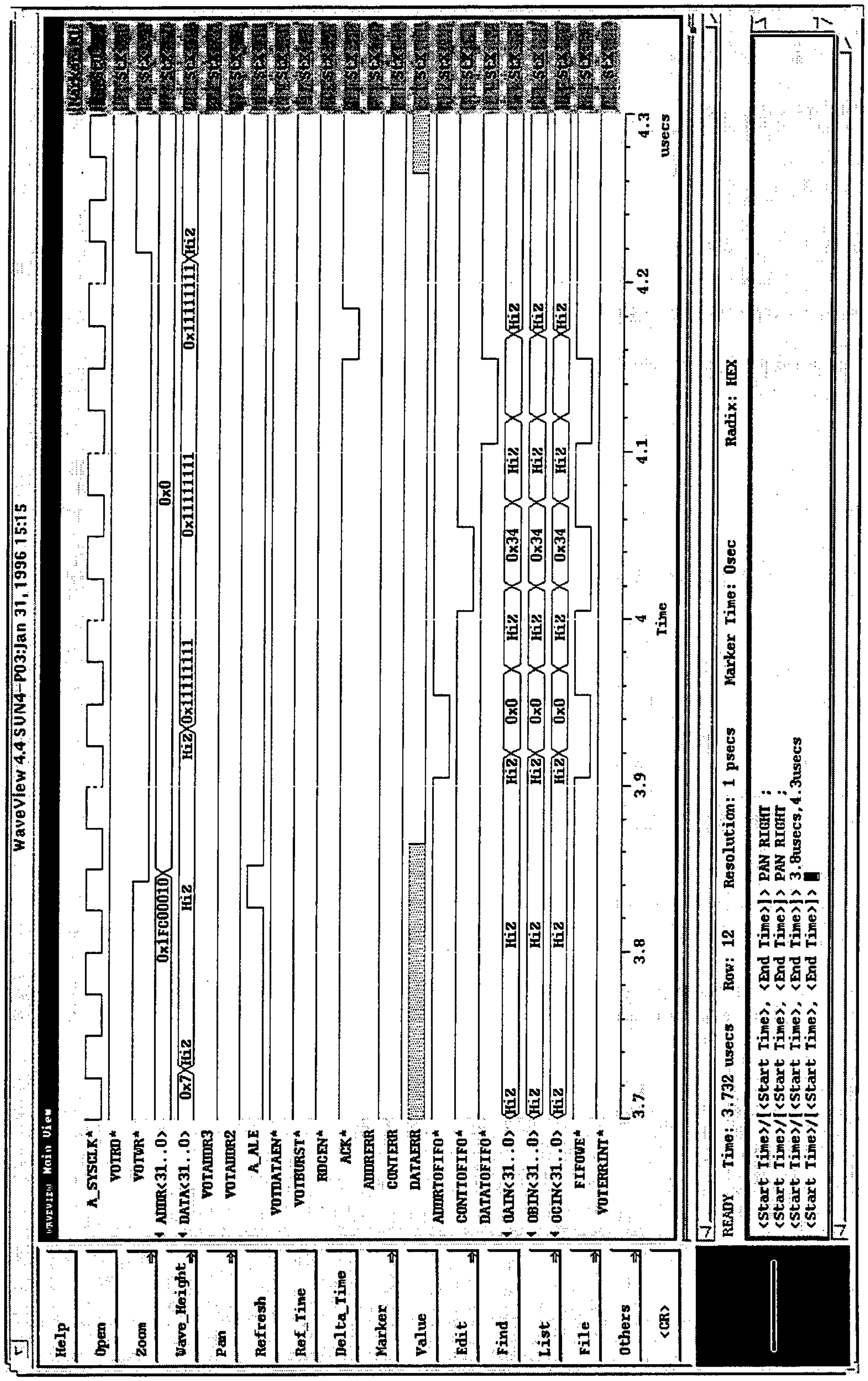
The vote error interrupt, ***VOTERRINT****, is generated only at the end of the current bus cycle. This allows the current bus cycle to complete, with the majority voters masking the associated fault. In addition, allowing the bus cycle to complete ensures the FIFOs associated with each processor capture the state of the address, control, and data bus of each processor prior to generating an interrupt.

It is intended that the three processors will synchronously receive the interrupt, and will execute the same interrupt service routine. The beginning and end of this service routine is indicated by a write to "dummy" address $1F80xxxx_H$. This address is decoded by the memory decoder to generate the chip select ***INTCS****. The error cycle controller, upon detection of a write cycle with this chip



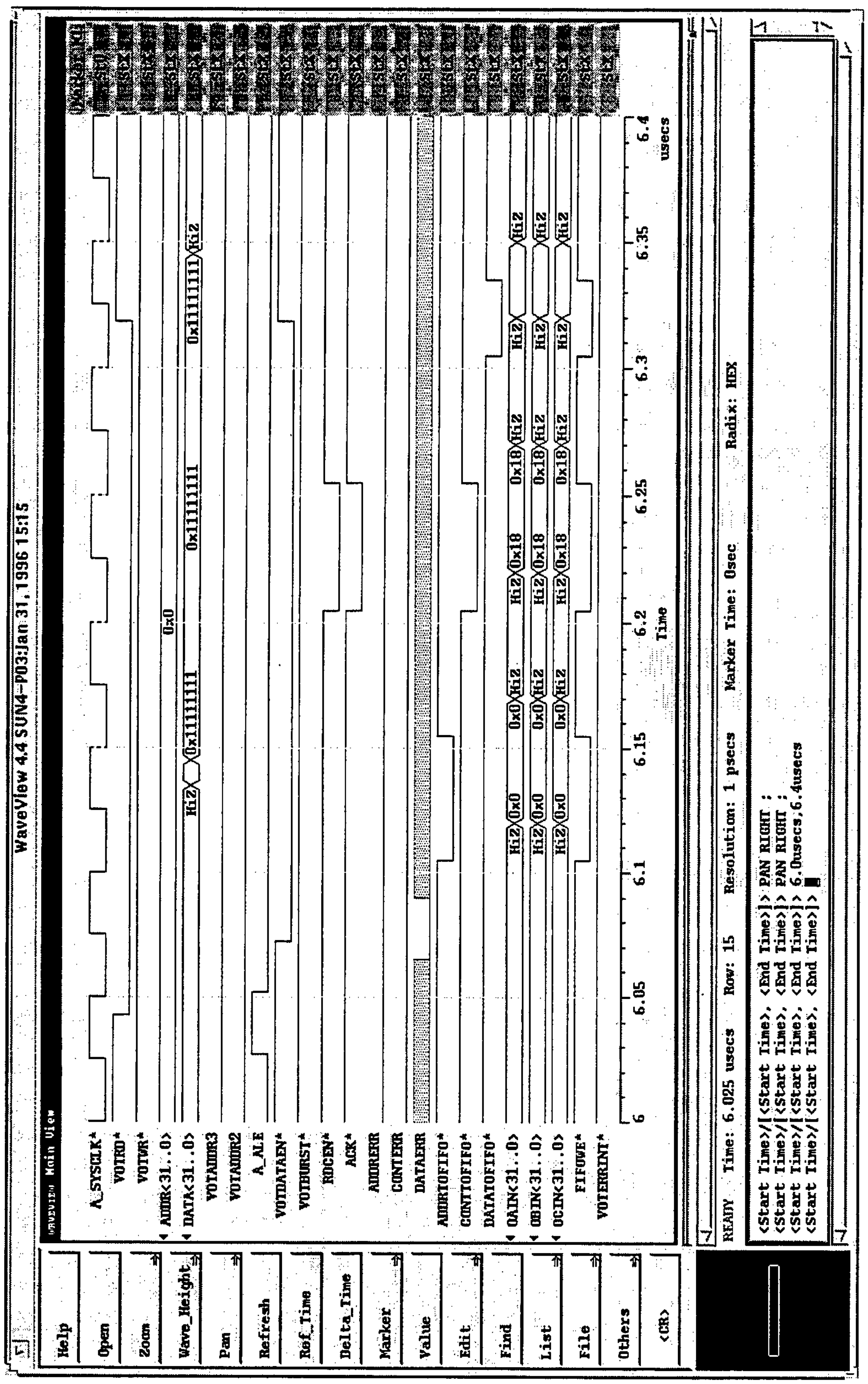
```
READY  Time: 1.023 usecs   Row: 22   Resolution: 1 psecs   Marker Time: 0sec   Radix: HEX
<Start Time>/[<Start Time>, <End Time>]> 0.8usecs,2.4usecs
<Start Time>/[<Start Time>, <End Time>]> 1.0usecs,2.4usecs
<Start Time>/[<Start Time>, <End Time>]> 1.1usecs,2.4usecs
<Start Time>/[<Start Time>, <End Time>]> █
```

Figure 26. FIFO Controls During Burst Read Cycle.



```
READY   Time: 3.732 usecs   Row: 12   Resolution: 1 psecs   Marker Time: 0sec   Radix: HEX
<Start Time>/[<Start Time>, <End Time>]> PAN RIGHT ;
<Start Time>/[<Start Time>, <End Time>]> PAN RIGHT ;
<Start Time>/[<Start Time>, <End Time>]> 3.8usecs,4.3usecs
<Start Time>/[<Start Time>, <End Time>]>
```

Figure 27. FIFO Controls During Write Cycle.



```
READY   Time: 6.025 usecs   Row: 15   Resolution: 1 psecs   Marker Time: 0sec   Radix: HEX
<Start Time>/[<Start Time>, <End Time>]> PAN RIGHT ;
<Start Time>/[<Start Time>, <End Time>]> PAN RIGHT ;
<Start Time>/[<Start Time>, <End Time>]> 6.0usecs,6.4usecs
<Start Time>/[<Start Time>, <End Time>]> █
```

Figure 28. FIFO Controls During Read Cycle.

select asserted, clears the interrupt and disables further vote error interrupts. The interrupt is disabled until the end of the interrupt routine. This is again signaled by the next write to "dummy" address 1F80xxxx$_H$.

During the interrupt routine, it is intended that the processors will write all of their internal general purpose registers, configuration registers, and instruction and data caches to some selected portion of RAM. The vote error interrupt will have been disabled. However, errors in the "faulty" processor will be masked by the majority voted output from the other two "agreeing" processors during each write. Then, the interrupt routine would read back the selected portion of RAM and refill all of its internal general purpose registers, configuration registers, and instruction and data caches. Thus, the processor which had an error will have been corrected and re-synchronized with the other two processors. While this routine is executing, the FIFOs associated with each processor will capture all of the internal information of each processor for error analysis.

The IDT R3081 Microprocessor Bus Interface Simulator module contained in Appendix A, Section A, contains a simulated, abbreviated interrupt service routine which executes when the interrupt ***INT5**** is asserted. Simulations which show the operation of the error cycle and this simulated interrupt service routine are contained in Chapter V.

I. SYSTEM INTERFACE

The system interface is intended to be a laptop or similar system which can read the FIFOs associated with each microprocessor and perform some analysis. This provides for both real-time and post error analysis. The FIFOs selected allow for asynchronous writing and reading with separate write and read clocks which can be different frequencies. Figure 29 is a block diagram of the system interface.

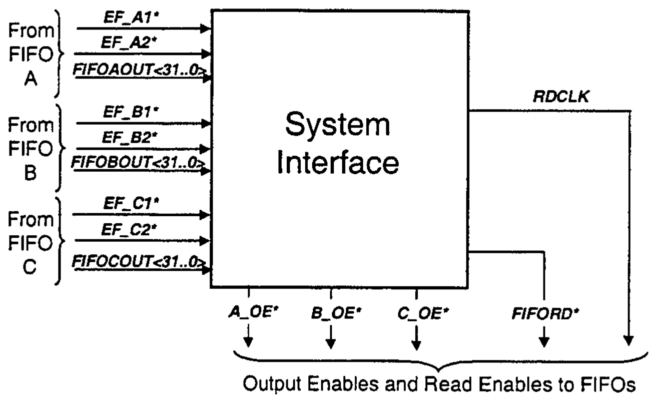
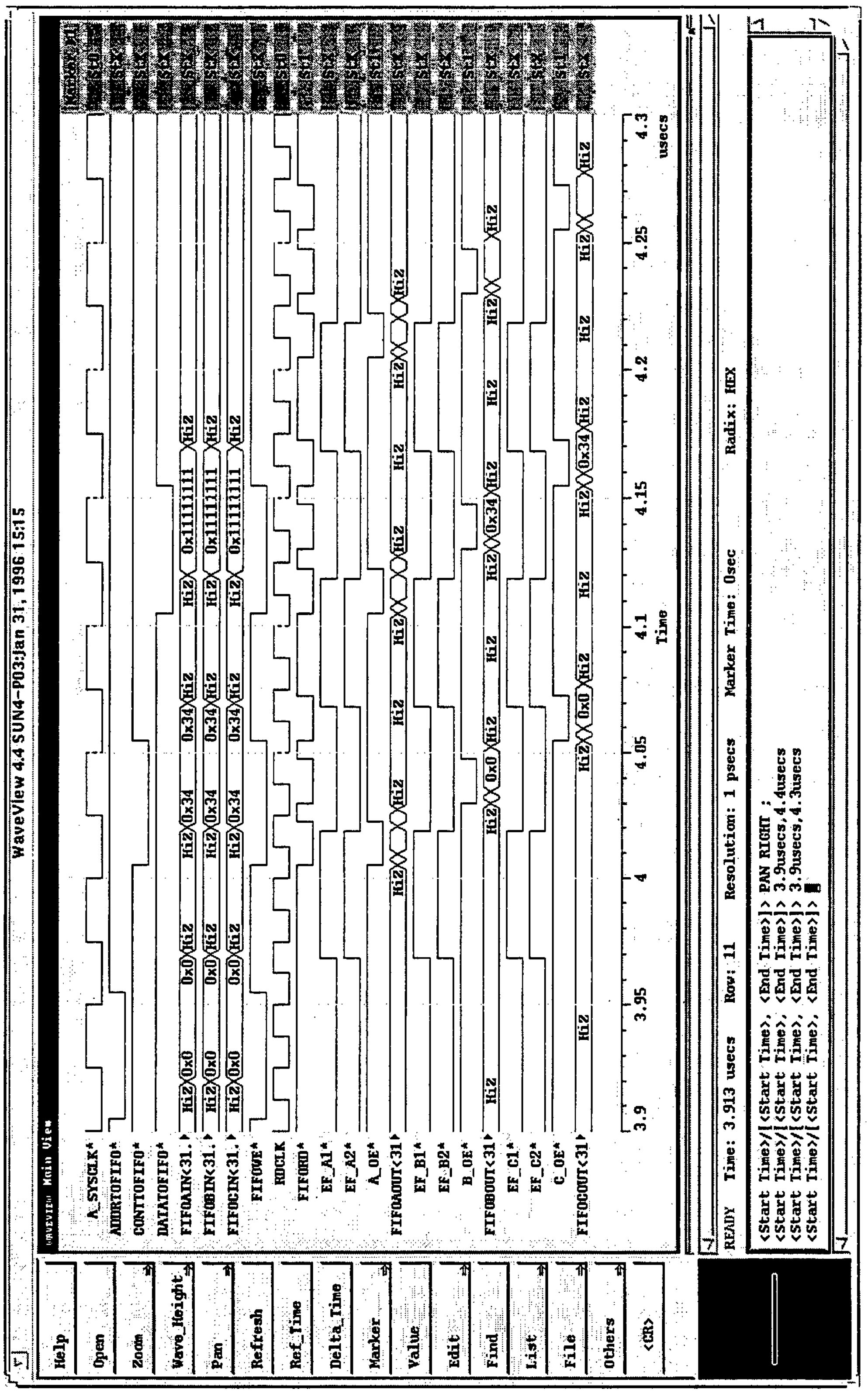


Figure 29. System Interface.

The testbed interface monitors the FIFO empty lines from processor A's FIFO, ***EF_A1**** and ***EF_A2****. As soon as they are both deasserted, the interface reads the FIFO. This is followed by monitoring the FIFO empty lines from processor B's FIFO, ***EF_B1**** and ***EF_B2*,*** and reading processor B's FIFO once they are both deasserted. Finally, the FIFO empty lines from processor C's FIFO, ***EF_C1**** and ***EF_C2*,*** are monitored and the FIFO is read once they are both deasserted. This process continues and the address, control, and data information stored in the associated FIFOs are obtained by the interface. The read clock is set to be twice the frequency of the write clock. This enables the interface to read the data out of the FIFOs fast enough so they never fill up. Figure 30 shows the timing of the control signals generated by the system interface.

The interface module writes the results obtained from the FIFOs to a text file, TMR_trace.out. By reviewing this text file, the status of the processors during each bus cycle can be observed. Examples of this text file obtained during both normal (error free) and induced error operations are contained in Chapter V.

The schematics for the system interface and associated Verilog code are contained in Appendix C, Section J.



```
READY   Time: 3.913 usecs   Row: 11   Resolution: 1 psecs   Marker Time: 0sec   Radix: HEX
<Start Time>/[<Start Time>, <End Time>]> PAN RIGHT ;
<Start Time>/[<Start Time>, <End Time>]> 3.9usecs,4.4usecs
<Start Time>/[<Start Time>, <End Time>]> 3.9usecs,4.3usecs
<Start Time>/[<Start Time>, <End Time>]>
```

Figure 30. System Interface Controls.

V. SIMULATION RESULTS

The complete design has been implemented in Cadence Concept™ schematics and the Verilog® Hardware Description Language. Timing parameters have been obtained from actual device datasheets. The IDT R3081 bus/memory interface in this TMR design can be simulated in Cadence Logic Workbench™ to verify the concept of operation and test the voting logic, memory and error cycle controllers, as well as the FIFO interface.

The following simulation results were obtained from the trace file generated by the simulated system interface. The information displayed represents what was actually read from each FIFO.

The overall testbed schematics are contained in Appendix A. The Cadence supplied modules and user defined modules used in the schematics and the simulations are contained in Appendices B and C, respectively. The script control language (SCL) files which were used to drive the inputs to the Testbed schematics to obtain the following simulation results are contained in Appendix D.

A. NORMAL (ERROR FREE) RESULTS

Bus cycles 1 through 4 correspond to a Burst Read from EPROM addresses 1FC00000$_H$ through 1FC0000C$_H$. The data read corresponds to the data programmed into the Verilog EPROM module in Appendix C, Section I.

```
                    CPU A          CPU B        CPU C
         =================================================
1.       Address = 1fc00000       1fc00000     1fc00000
         Control = 00000008       00000008     00000008
         Data    = 00000000       00000000     00000000
         A Control = Burst Read Word 0
         B Control = Burst Read Word 0
         C Control = Burst Read Word 0
         =================================================
2.       Address = 1fc00000       1fc00000     1fc00000
         Control = 00000009       00000009     00000009
         Data    = 00000001       00000001     00000001
         A Control = Burst Read Word 1
         B Control = Burst Read Word 1
         C Control = Burst Read Word 1
         =================================================
3.       Address = 1fc00000       1fc00000     1fc00000
         Control = 0000000a       0000000a     0000000a
         Data    = 00000002       00000002     00000002
         A Control = Burst Read Word 2
         B Control = Burst Read Word 2
         C Control = Burst Read Word 2
         =================================================
4.       Address = 1fc00000       1fc00000     1fc00000
         Control = 0000000b       0000000b     0000000b
         Data    = 00000003       00000003     00000003
         A Control = Burst Read Word 3
         B Control = Burst Read Word 3
         C Control = Burst Read Word 3
         =================================================
```

Bus cycles 5 through 8 correspond to a Burst Read from EPROM addresses 1FC00010$_H$ through 1FC0001C$_H$. Again the data read corresponds to the data programmed into the Verilog EPROM module in Appendix C, Section I.

```
5.     Address = 1fc00010      1fc00010    1fc00010
       Control = 00000008      00000008    00000008
       Data    = 00000004      00000004    00000004
       A Control = Burst Read Word 0
       B Control = Burst Read Word 0
       C Control = Burst Read Word 0
       ================================================
6.     Address = 1fc00010      1fc00010    1fc00010
       Control = 00000009      00000009    00000009
       Data    = 00000005      00000005    00000005
       A Control = Burst Read Word 1
       B Control = Burst Read Word 1
       C Control = Burst Read Word 1
       ================================================
7.     Address = 1fc00010      1fc00010    1fc00010
       Control = 0000000a      0000000a    0000000a
       Data    = 00000006      00000006    00000006
       A Control = Burst Read Word 2
       B Control = Burst Read Word 2
       C Control = Burst Read Word 2
       ================================================
8.     Address = 1fc00010      1fc00010    1fc00010
       Control = 0000000b      0000000b    0000000b
       Data    = 00000007      00000007    00000007
       A Control = Burst Read Word 3
       B Control = Burst Read Word 3
       C Control = Burst Read Word 3
       ================================================
```

Bus cycles 9 through 12 correspond to four Write bus cycles to RAM addresses 00000000_H, 00000004_H, 00000008_H, and $0000000C_H$.

```
9.     Address = 00000000      00000000    00000000
       Control = 00000034      00000034    00000034
       Data    = 11111111      11111111    11111111
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
10.    Address = 00000000      00000000    00000000
       Control = 00000035      00000035    00000035
       Data    = 22222222      22222222    22222222
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
11.    Address = 00000000      00000000    00000000
       Control = 00000036      00000036    00000036
       Data    = 33333333      33333333    33333333
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
12.   Address = 00000000     00000000    00000000
      Control = 00000037     00000037    00000037
      Data    = 44444444     44444444    44444444
      A Control = Write
      B Control = Write
      C Control = Write
      ================================================
```

Bus cycle 13 corresponds to a single word Read bus cycle from RAM address 00000000_H. The data read is the same that was written during cycle 9.

```
13.   Address = 00000000     00000000    00000000
      Control = 00000018     00000018    00000018
      Data    = 11111111     11111111    11111111
      A Control = Read
      B Control = Read
      C Control = Read
      ================================================
```

Bus cycles 14 through 17 correspond to a Burst Read from RAM addresses 00000000_H through $0000000C_H$. The data read from RAM is the same that was written during cycles 9 through 12.

```
14.   Address = 00000000     00000000    00000000
      Control = 00000008     00000008    00000008
      Data    = 11111111     11111111    11111111
      A Control = Burst Read Word 0
      B Control = Burst Read Word 0
      C Control = Burst Read Word 0
      ================================================
15.   Address = 00000000     00000000    00000000
      Control = 00000009     00000009    00000009
      Data    = 22222222     22222222    22222222
      A Control = Burst Read Word 1
      B Control = Burst Read Word 1
      C Control = Burst Read Word 1
      ================================================
16.   Address = 00000000     00000000    00000000
      Control = 0000000a     0000000a    0000000a
      Data    = 33333333     33333333    33333333
      A Control = Burst Read Word 2
      B Control = Burst Read Word 2
      C Control = Burst Read Word 2
      ================================================
17.  Address = 00000000      00000000     00000000
     Control = 0000000b      0000000b     0000000b
     Data    = 44444444      44444444     44444444
     A Control = Burst Read Word 3
     B Control = Burst Read Word 3
     C Control = Burst Read Word 3
     ================================================
```

B. INJECTED ERROR RESULTS

Bus cycles 1 through 4 correspond to a Burst Read from EPROM addresses $1FC00000_H$ through $1FC0000C_H$. The data read corresponds to the data programmed into the Verilog© EPROM module in Appendix C, Section I.

```
                 CPU A          CPU B           CPU C
     ================================================
1.   Address = 1fc00000      1fc00000     1fc00000
     Control = 00000008      00000008     00000008
     Data    = 00000000      00000000     00000000
     A Control = Burst Read Word 0
     B Control = Burst Read Word 0
     C Control = Burst Read Word 0
     ================================================
2.   Address = 1fc00000      1fc00000     1fc00000
     Control = 00000009      00000009     00000009
     Data    = 00000001      00000001     00000001
     A Control = Burst Read Word 1
     B Control = Burst Read Word 1
     C Control = Burst Read Word 1
     ================================================
3.   Address = 1fc00000      1fc00000     1fc00000
     Control = 0000000a      0000000a     0000000a
     Data    = 00000002      00000002     00000002
     A Control = Burst Read Word 2
     B Control = Burst Read Word 2
     C Control = Burst Read Word 2
     ================================================
4.   Address = 1fc00000      1fc00000     1fc00000
     Control = 0000000b      0000000b     0000000b
     Data    = 00000003      00000003     00000003
     A Control = Burst Read Word 3
     B Control = Burst Read Word 3
     C Control = Burst Read Word 3
     ================================================
```

Cycle 5 is a Write bus cycle to RAM address 00000000_H where there is an error in the address of processor A.

```
5.     Address = 00000100      00000000    00000000
       Control = 00000034      00000034    00000034
       Data    = 11111111      11111111    11111111
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
```

Cycles 6 through 11 are the six cycles of the simulated interrupt service routine. The differences between the "internal" information of the three processors that caused the error can be observed. These differences do not themselves cause additional vote error interrupts because the interrupt routines are initiated by a write to "dummy" address 1F80xxxx$_H$. However, when the "internal" information is read back from RAM, the "corrected" information is read.

```
6.     Address = 1f800000      1f800000    1f800000
       Control = 00000034      00000034    00000034
       Data    = ffffffff      ffffffff    ffffffff
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
7.     Address = 00070000      00070000    00070000
       Control = 00000034      00000034    00000034
       Data    = 00000100      00000000    00000000
       A Control = Write
       B Control = Write
       C Control = Write
8.     ================================================
       Address = 00070000      00070000    00070000
       Control = 00000035      00000035    00000035
       Data    = 11111111      11111111    11111111
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
9.     Address = 00070000      00070000    00070000
       Control = 00000018      00000018    00000018
       Data    = 00000000      00000000    00000000
       A Control = Read
       B Control = Read
       C Control = Read
       ================================================
10.     Address = 00070000      00070000    00070000
        Control = 00000019      00000019    00000019
        Data    = 11111111      11111111    11111111
        A Control = Read
        B Control = Read
        C Control = Read
        ==================================================
11.     Address = 1f800000      1f800000    1f800000
        Control = 00000034      00000034    00000034
        Data    = ffffffff      ffffffff    ffffffff
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
```

Cycle 12 is a Write bus cycle to RAM address 00000004_H where there is an error in the address of processor B. Cycles 13 through 18 are the simulated interrupt service routine initiated by the three processors.

```
12.     Address = 00000000      01000000    00000000
        Control = 00000035      00000035    00000035
        Data    = 22222222      22222222    22222222
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
13.     Address = 1f800000      1f800000    1f800000
        Control = 00000034      00000034    00000034
        Data    = ffffffff      ffffffff    ffffffff
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
14.     Address = 00070000      00070000    00070000
        Control = 00000034      00000034    00000034
        Data    = 00000004      01000004    00000005
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
15.     Address = 00070000      00070000    00070000
        Control = 00000035      00000035    00000035
        Data    = 22222222      22222222    22222222
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
16.     Address = 00070000      00070000    00070000
        Control = 00000018      00000018    00000018
        Data    = 00000004      00000004    00000004
        A Control = Read
        B Control = Read
        C Control = Read
        ==================================================
17.     Address = 00070000      00070000    00070000
        Control = 00000019      00000019    00000019
        Data    = 22222222      22222222    22222222
        A Control = Read
        B Control = Read
        C Control = Read
        ===============================================
18.     Address = 1f800000      1f800000    1f800000
        Control = 00000034      00000034    00000034
        Data    = ffffffff      ffffffff    ffffffff
        A Control = Write
        B Control = Write
        C Control = Write
        ===============================================
```

Cycle 19 is a Write bus cycle to RAM address 00000008_H where there is an error in the data of processor C. Cycles 20 through 25 are the simulated interrupt service routine initiated by the three processors.

```
19.     Address = 00000000      00000000    00000000
        Control = 00000036      00000036    00000036
        Data    = 33333333      33333333    33333337
        A Control = Write
        B Control = Write
        C Control = Write
        ===============================================
20.     Address = 1f800000      1f800000    1f800000
        Control = 00000034      00000034    00000034
        Data    = ffffffff      ffffffff    ffffffff
        A Control = Write
        B Control = Write
        C Control = Write
        ===============================================
21.     Address = 00070000      00070000    00070000
        Control = 00000034      00000034    00000034
        Data    = 00000008      00000008    00000008
        A Control = Write
        B Control = Write
        C Control = Write
        ===============================================
22.     Address = 00070000      00070000    00070000
        Control = 00000035      00000035    00000035
        Data    = 33333333      33333333    33333337
        A Control = Write
        B Control = Write
        C Control = Write
        ===============================================
23.     Address = 00070000      00070000    00070000
        Control = 00000018      00000018    00000018
        Data    = 00000008      00000008    00000008
        A Control = Read
        B Control = Read
        C Control = Read
        ===============================================
24.     Address = 00070000       00070000     00070000
        Control = 00000019       00000019     00000019
        Data    = 33333333       33333333     33333333
        A Control = Read
        B Control = Read
        C Control = Read
        ==================================================
25.     Address = 1f800000       1f800000     1f800000
        Control = 00000034       00000034     00000034
        Data    = ffffffff       ffffffff     ffffffff
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
```

Cycle 26 is a Write bus cycle to RAM address $0000000A_H$ where there are multiple errors in the data of all three processors. Cycles 27 through 32 are the interrupt service routine.

```
26.     Address = 00000000       00000000     00000000
        Control = 00000037       00000037     00000037
        Data    = f4444444       44a44444     44444447
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
27.     Address = 1f800000       1f800000     1f800000
        Control = 00000034       00000034     00000034
        Data    = ffffffff       ffffffff     ffffffff
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
28.     Address = 00070000       00070000     00070000
        Control = 00000034       00000034     00000034
        Data    = 0000000c       0000000c     0000000c
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
29.     Address = 00070000       00070000     00070000
        Control = 00000035       00000035     00000035
        Data    = f4444444       44a44444     44444447
        A Control = Write
        B Control = Write
        C Control = Write
        ==================================================
30.     Address = 00070000       00070000     00070000
        Control = 00000018       00000018     00000018
        Data    = 0000000c       0000000c     0000000c
        A Control = Read
        B Control = Read
        C Control = Read
        ==================================================
31.   Address = 00070000      00070000    00070000
      Control = 00000019      00000019    00000019
      Data    = 44444444      44444444    44444444
      A Control = Read
      B Control = Read
      C Control = Read
      ==================================================
32.   Address = 1f800000      1f800000    1f800000
      Control = 0000034       0000034     0000034
      Data    = ffffffff      ffffffff    ffffffff
      A Control = Write
      B Control = Write
      C Control = Write
      ==================================================
```

Cycles 33 through 36 are a Burst Read from RAM addresses 00000000_H, 00000004_H, 00000008_H, and $0000000C_H$. The data read from RAM is the data which was "corrected" by the majority voter when written during cycles 5, 12, 19, and 26. This example shows the successful completion of the four Write cycles (5, 12, 19, and 26) which contained errors.

```
33.   Address = 00000000      00000000    00000000
      Control = 00000008      00000008    00000008
      Data    = 11111111      11111111    11111111
      A Control = Burst Read Word 0
      B Control = Burst Read Word 0
      C Control = Burst Read Word 0
      ==================================================
34.   Address = 00000000      00000000    00000000
      Control = 00000009      00000009    00000009
      Data    = 22222222      22222222    22222222
      A Control = Burst Read Word 1
      B Control = Burst Read Word 1
      C Control = Burst Read Word 1
      ==================================================
35.   Address = 00000000      00000000    00000000
      Control = 0000000a      0000000a    0000000a
      Data    = 33333333      33333333    33333333
      A Control = Burst Read Word 2
      B Control = Burst Read Word 2
      C Control = Burst Read Word 2
      ==================================================
36.   Address = 00000000      00000000    00000000
      Control = 0000000b      0000000b    0000000b
      Data    = 44444444      44444444    44444444
      A Control = Burst Read Word 3
      B Control = Burst Read Word 3
      C Control = Burst Read Word 3
      ==================================================
```

Cycle 37 is a Write cycle to RAM address 00004000_H where processor B has incorrectly initiated a burst read from 00004000_H. Cycles 38 through 43 are the interrupt routine.

```
37.    Address = 00004000      00004000    00004000
       Control = 00000034      00000008    00000034
       Data    = 78787878      xxxxxxxx    78787878
       A Control = Write
       B Control = Burst Read Word 0
       C Control = Write
       ================================================
38.    Address = 1f800000      1f800000    1f800000
       Control = 00000034      00000034    00000034
       Data    = ffffffff      ffffffff    ffffffff
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
39.    Address = 00070000      00070000    00070000
       Control = 00000034      00000034    00000034
       Data    = 00004000      00004000    00004000
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
40.    Address = 00070000      00070000    00070000
       Control = 00000035      00000035    00000035
       Data    = 78787878      78787878    78787878
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
41.    Address = 00070000      00070000    00070000
       Control = 00000018      00000018    00000018
       Data    = 00004000      00004000    00004000
       A Control = Read
       B Control = Read
       C Control = Read
       ================================================
42.    Address = 00070000      00070000    00070000
       Control = 00000019      00000019    00000019
       Data    = 78787878      78787878    78787878
       A Control = Read
       B Control = Read
       C Control = Read
       ================================================
43.    Address = 1f800000      1f800000    1f800000
       Control = 00000034      00000034    00000034
       Data    = ffffffff      ffffffff    ffffffff
       A Control = Write
       B Control = Write
       C Control = Write
       ================================================
```

Cycle 44 is a single word Read from RAM address 00004000_H. The data read is the correct data written during cycle 37.

```
44.   Address = 00004000      00004000    00004000
      Control = 00000018      00000018    00000018
      Data    = 78787878      78787878    78787878
      A Control = Read
      B Control = Read
      C Control = Read
      ================================================
```

VI. CONCLUSION

With the rapidly declining radiation hardened device market and high prices of such devices when compared to COTS alternatives, a tool is desired that will allow the observance and analysis of COTS processors operating in a radiation environment. Additional reasons to move towards COTS devices are significant advantages in efficiency, performance, and software availability.

One of the primary disadvantages of COTS devices is their susceptibility to single event upsets. Triple Modular Redundancy (TMR) is viewed as one of many possible alternatives to provide some protection from SEUs in COTS devices.

The danger of incorporating redundancy into a system is that the overall system reliability could be reduced, due to the increased number of components. If the redundant systems are not themselves reliable, there is little hope of improving the reliability of the system.

The TMR Testbed design is not intended as a design for space flight operations. Nor is it intended as a guaranteed method of improving the performance of the R3081 processors in the presence of radiation induced single event upsets. The design herein is intended for ground based operational testing of the voting logic and any software algorithms run within the processors themselves. It is assumed that the board can be constructed in such a way that all of the hardware, other than the microprocessors, can be adequately shielded during laboratory radiation testing. In addition, it is realized that a fault which occurs in two of the processors at the same time, and which is manifested as the same bit being flipped on the address, control, or data bus, cannot be detected. In the event this error occurs, the two processors which are actually "faulty" will agree and become the majority when passed to the majority voters.

In the Testbed design, TMR provides the opportunity to monitor the three processors and in the event of an error, determine which processor was in error and what the processor was doing at the time the error occurred.

The Cadence/Verilog® design will allow simulation of the concept, verification of timing signals, and flexibility in reconfiguration of the design. Through simulation, the use of the bus/memory interface from three COTS microprocessors in a TMR design to monitor the system for errors has been realized. The actual board design could be constructed and used to test voting logic hardware and software algorithms in a laboratory environment in the presence of radiation induced SEUs or injected faults.

The use of the dedicated FIFO memories allows both real time and post-error analysis of the state of the three microprocessors. Thus, the tool will provide the capability to analyze the success or failure of attempts to improve the performance of COTS microprocessors in this environment, prior to their use in designs intended for actual space applications.

APPENDIX A. TMR TESTBED DESIGN SCHEMATICS

This appendix contains the entire schematic for the TMR Testbed built using Cadence Concept™ schematic tools and the Verilog® Hardware Description Language.

Enlarged views of each block in the following schematics and associated Verilog® code, when applicable, are contained Appendices B and C.

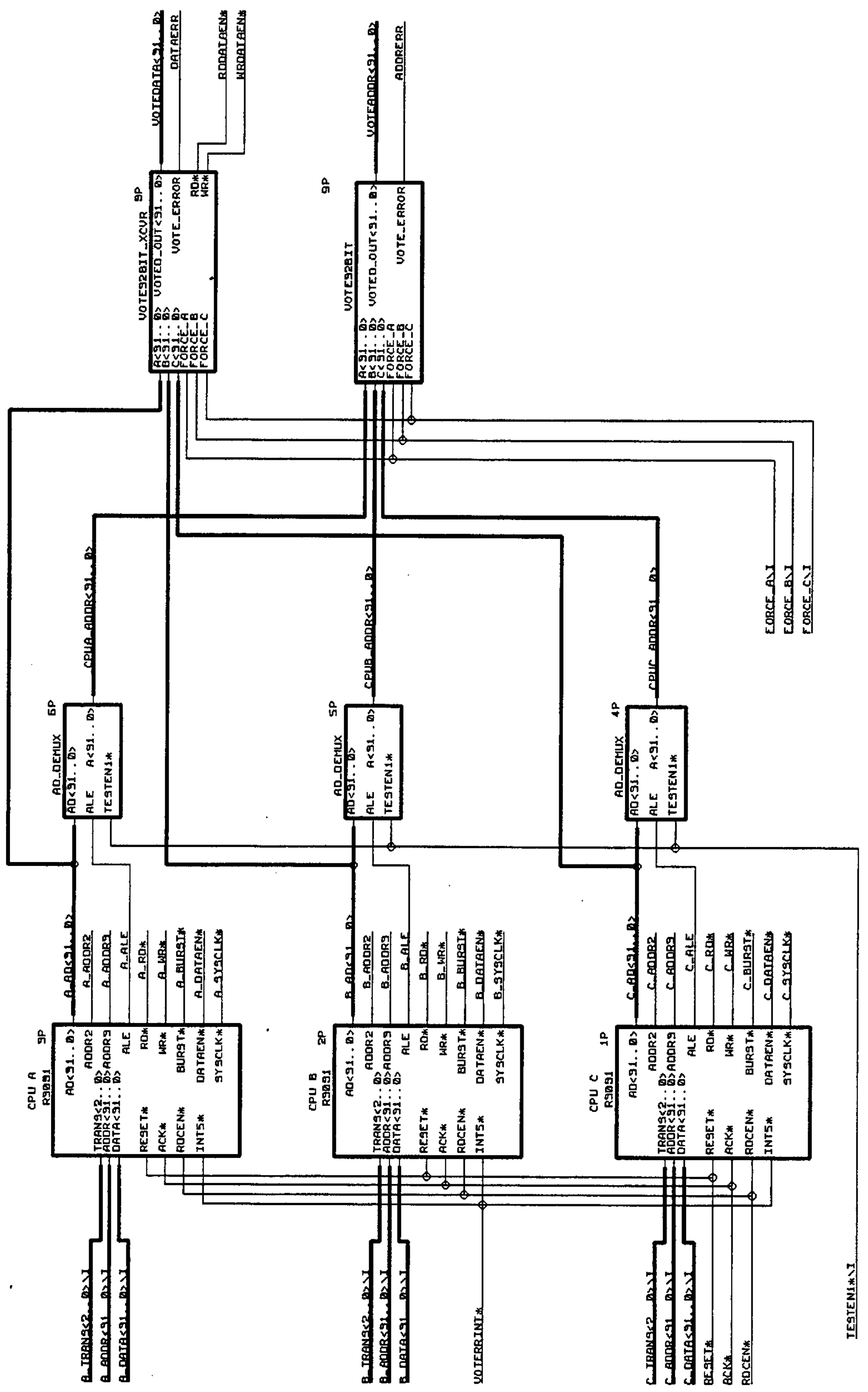


Figure 31. TMR Testbed Schematic (1 of 11).

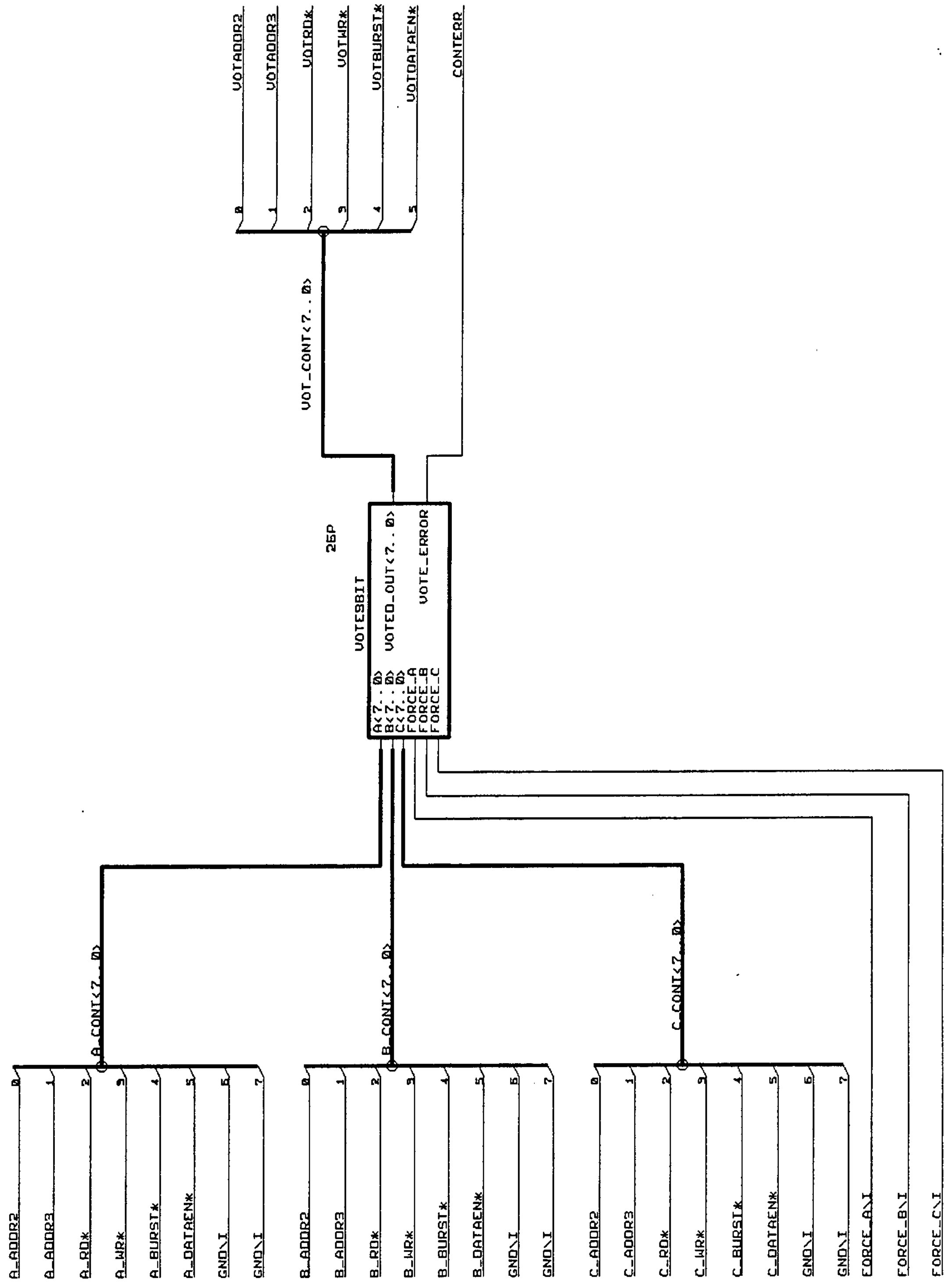


Figure 32. TMR Testbed Schematic (2 of 11).

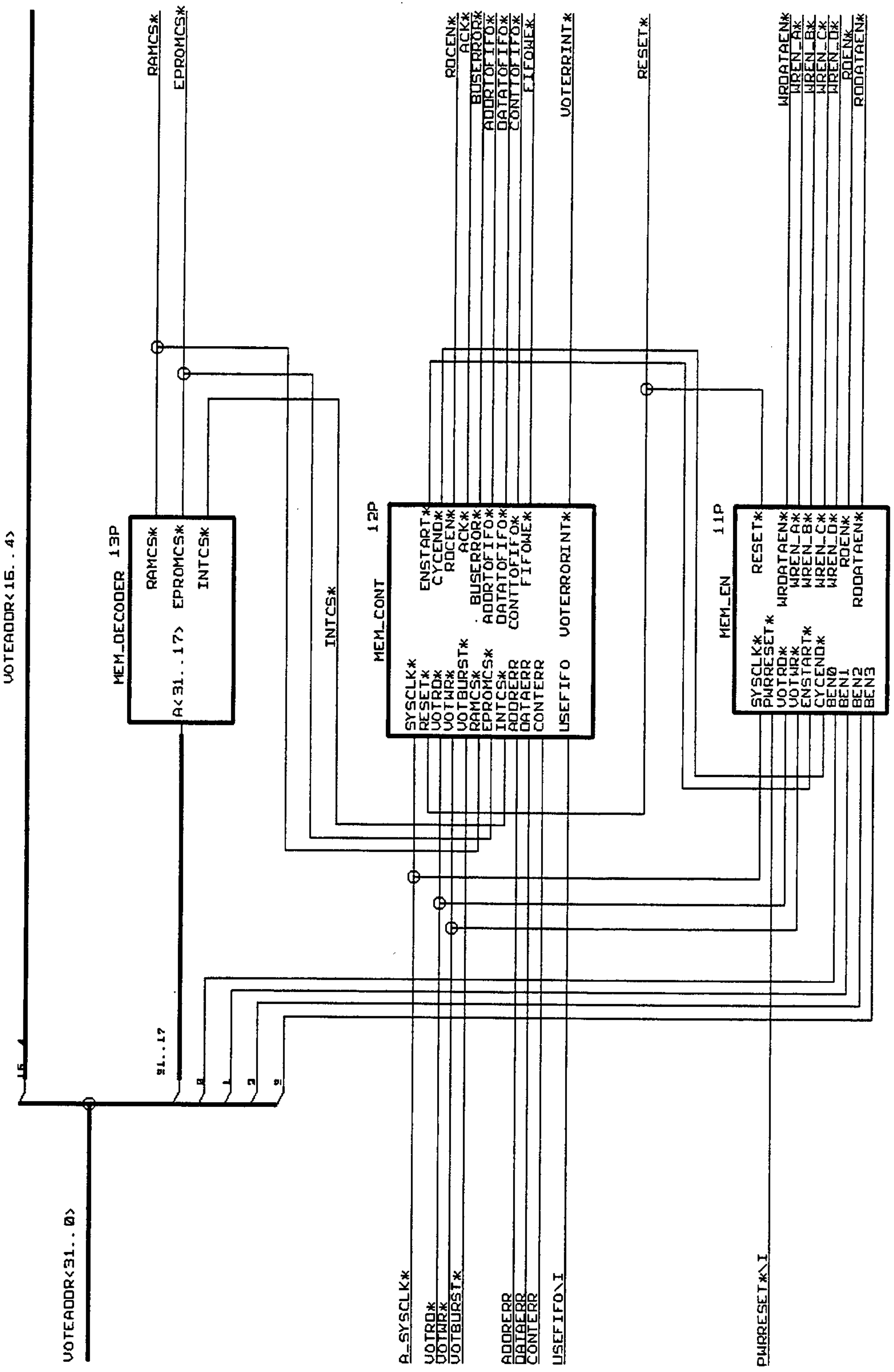


Figure 33. TMR Testbed Schematic (3 of 11).

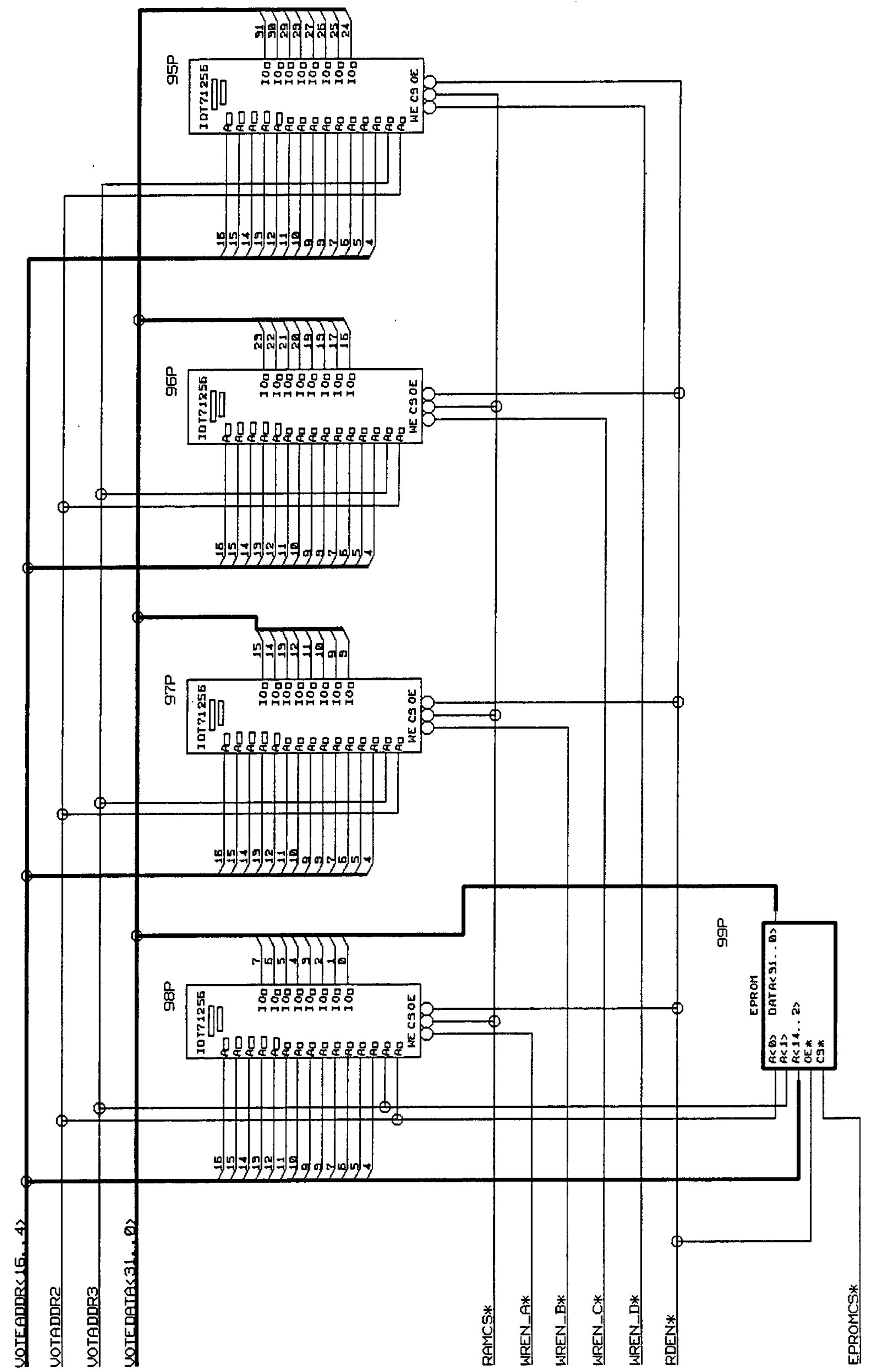


Figure 34. TMR Testbed Schematic (4 of 11).

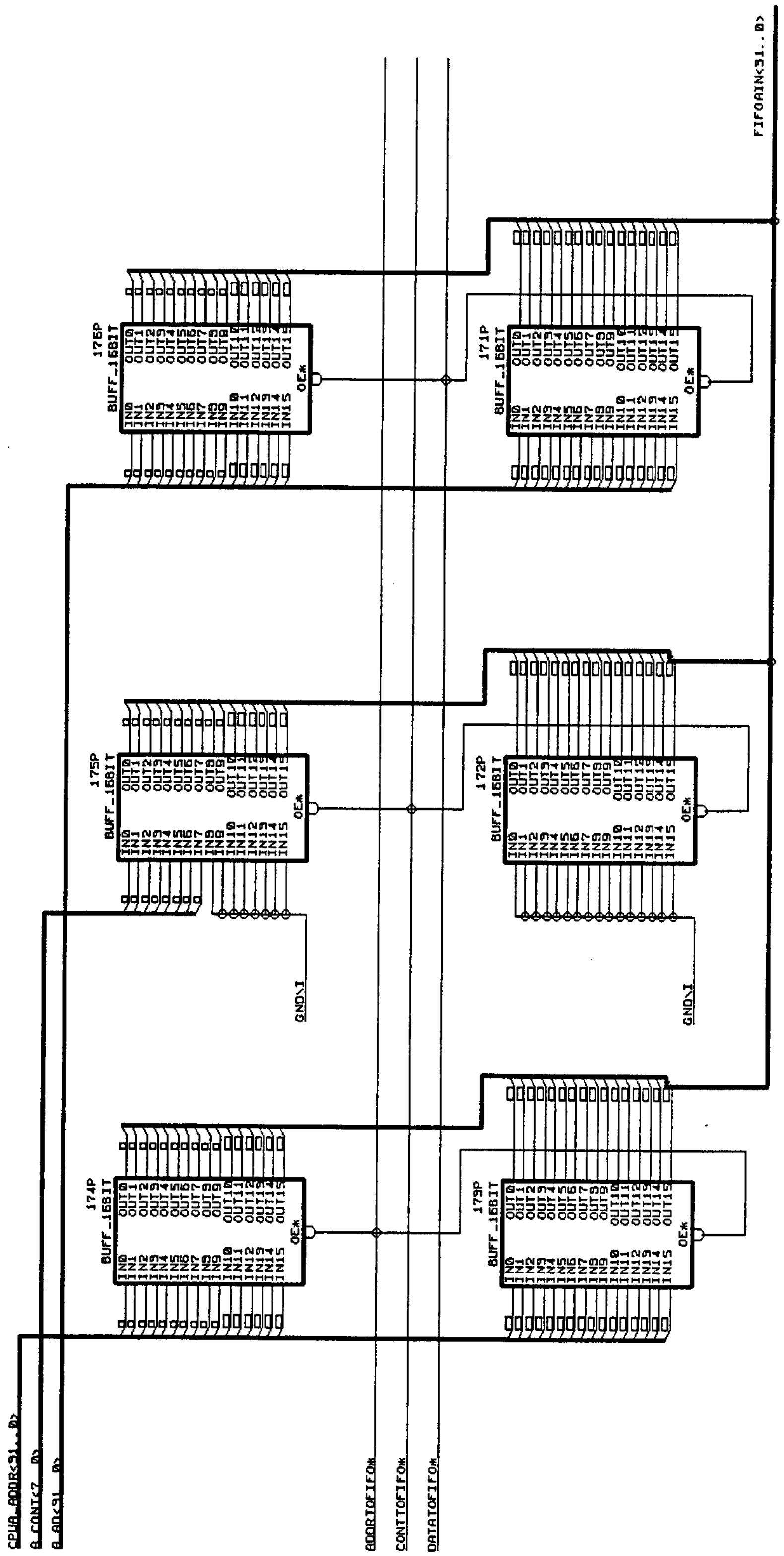


Figure 35. TMR Testbed Schematic (5 of 11).

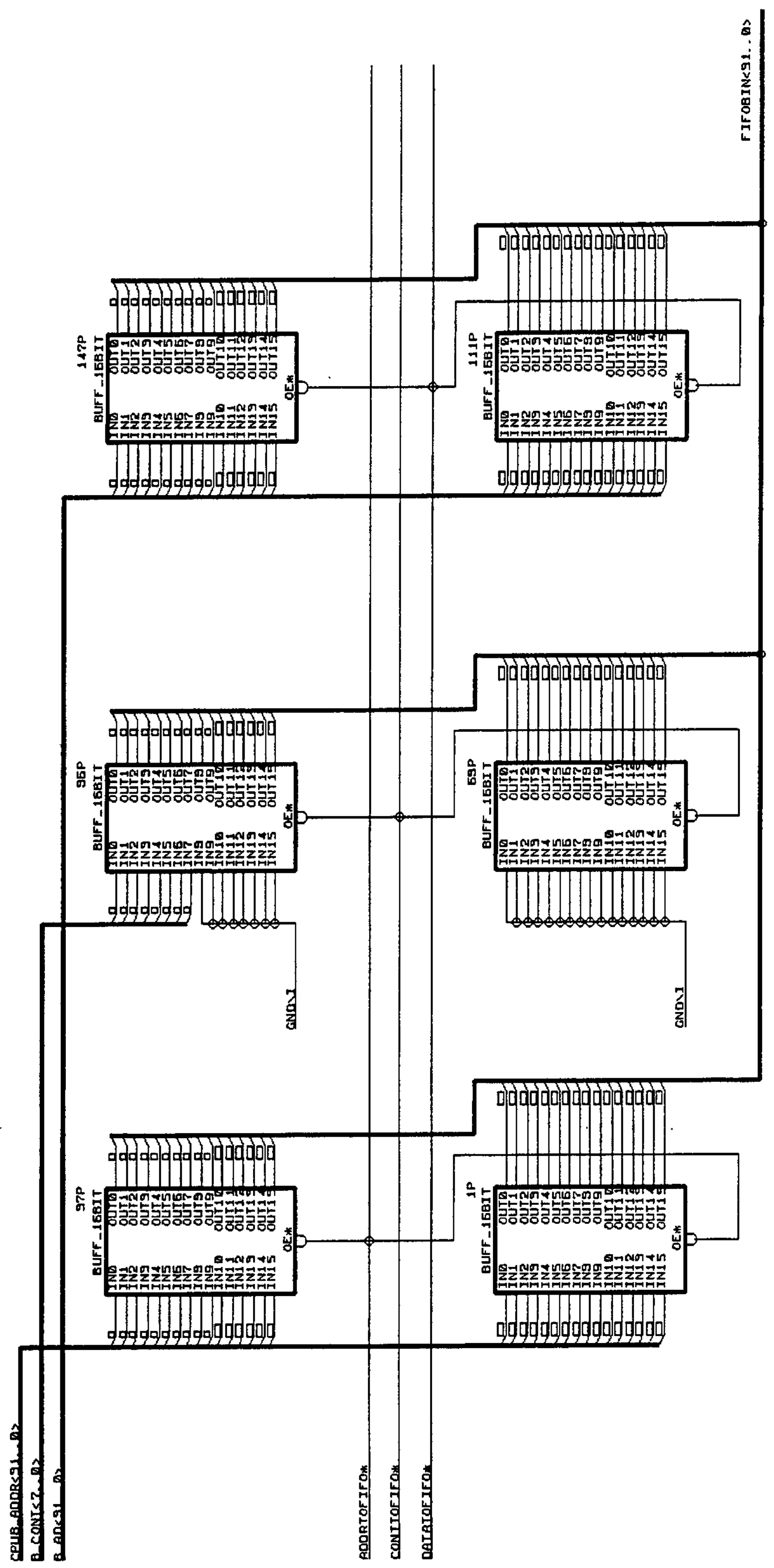


Figure 36. TMR Testbed Schematic (6 of 11).

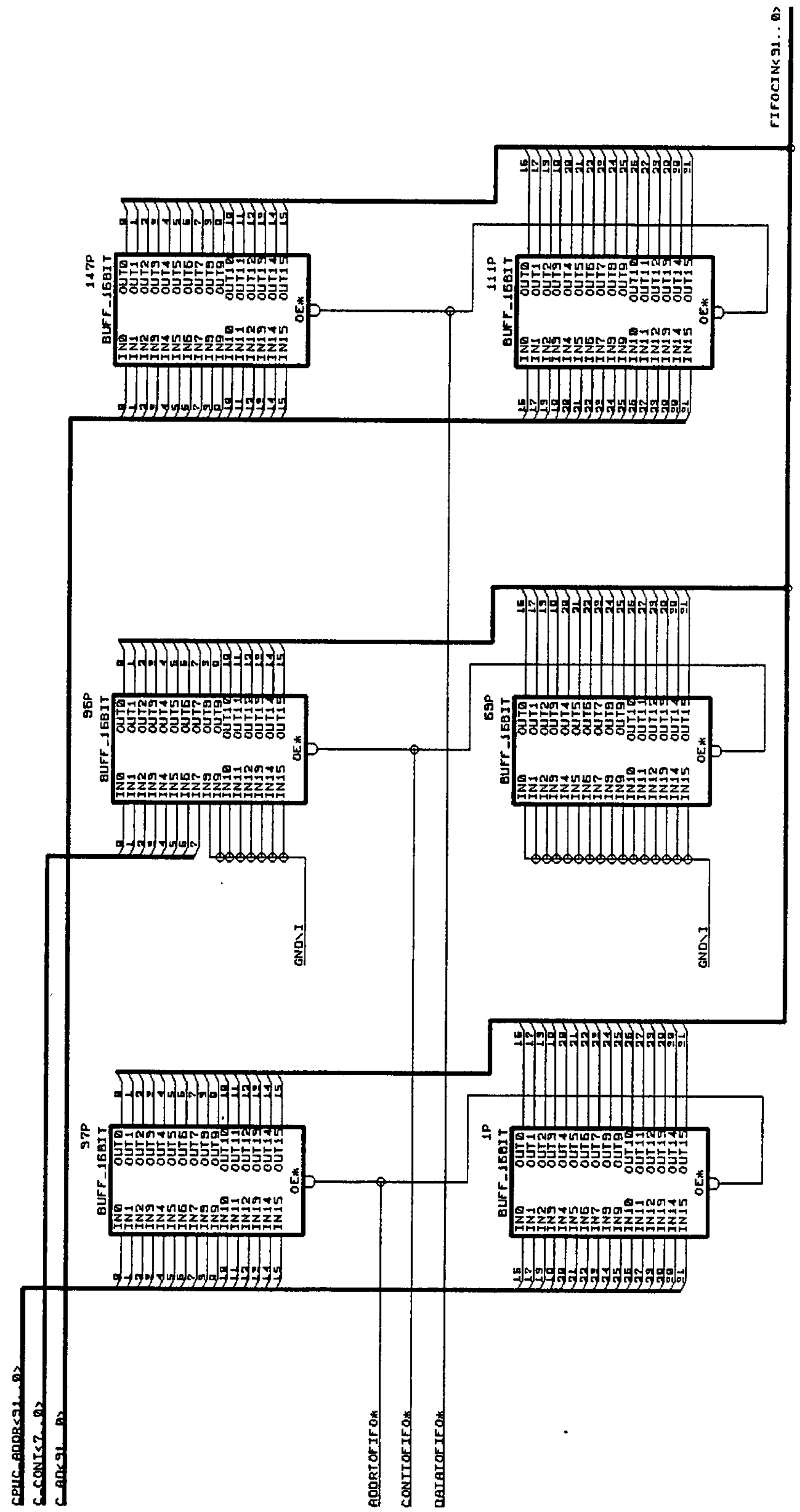


Figure 37. TMR Testbed Schematic (7 of 11).

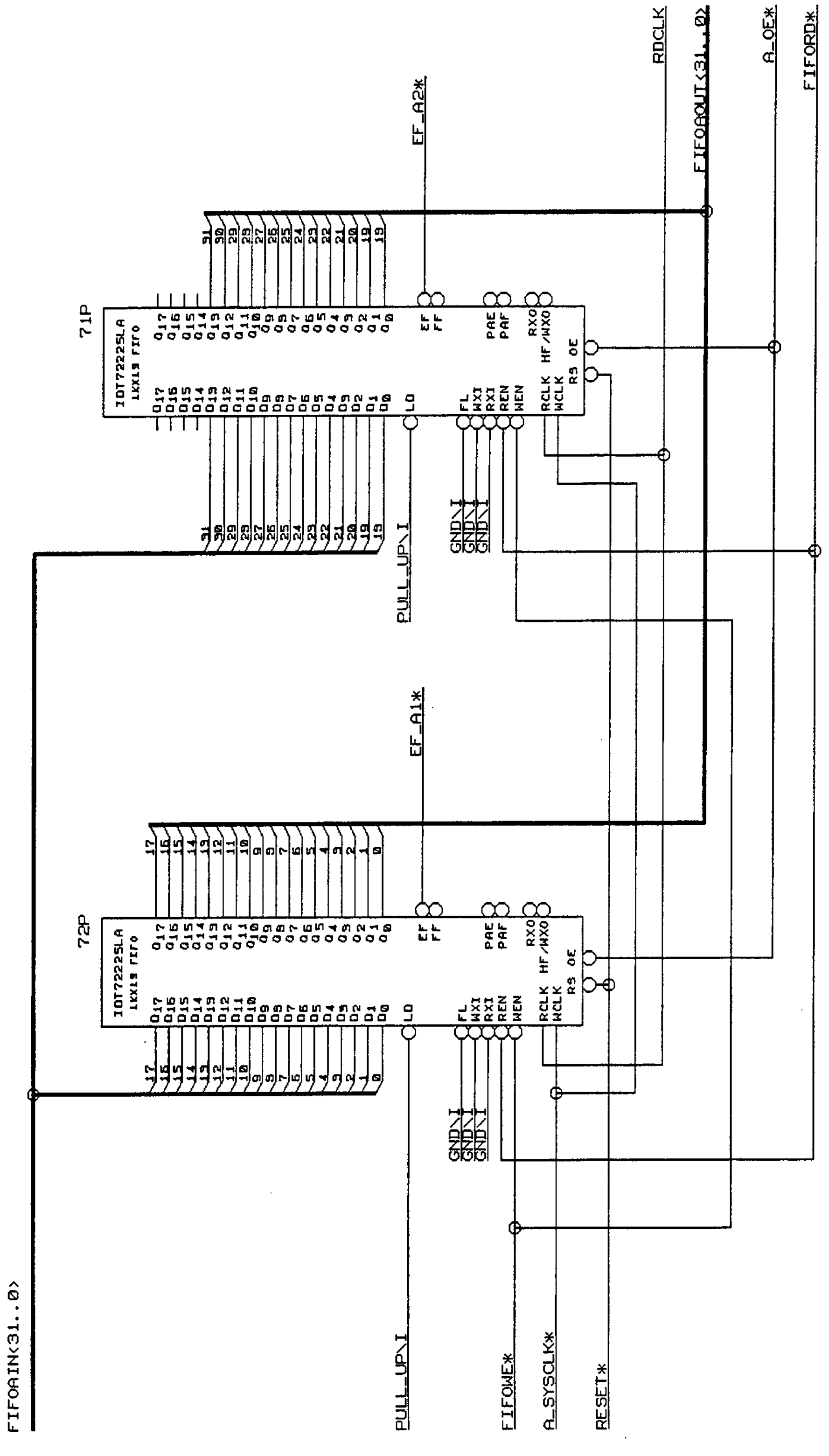


Figure 38. TMR Testbed Schematic (8 of 11).

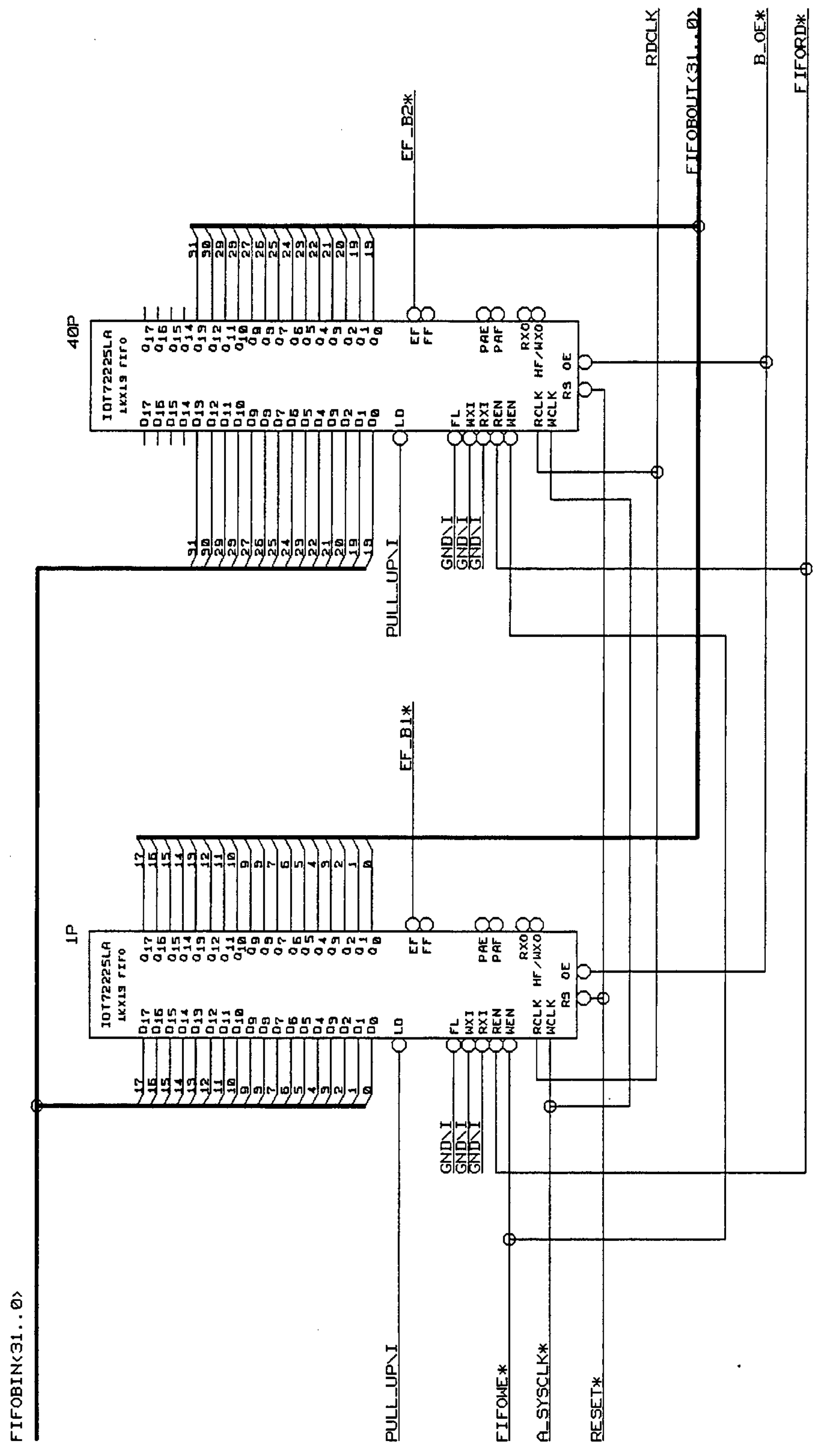


Figure 39. TMR Testbed Schematic (9 of 11).

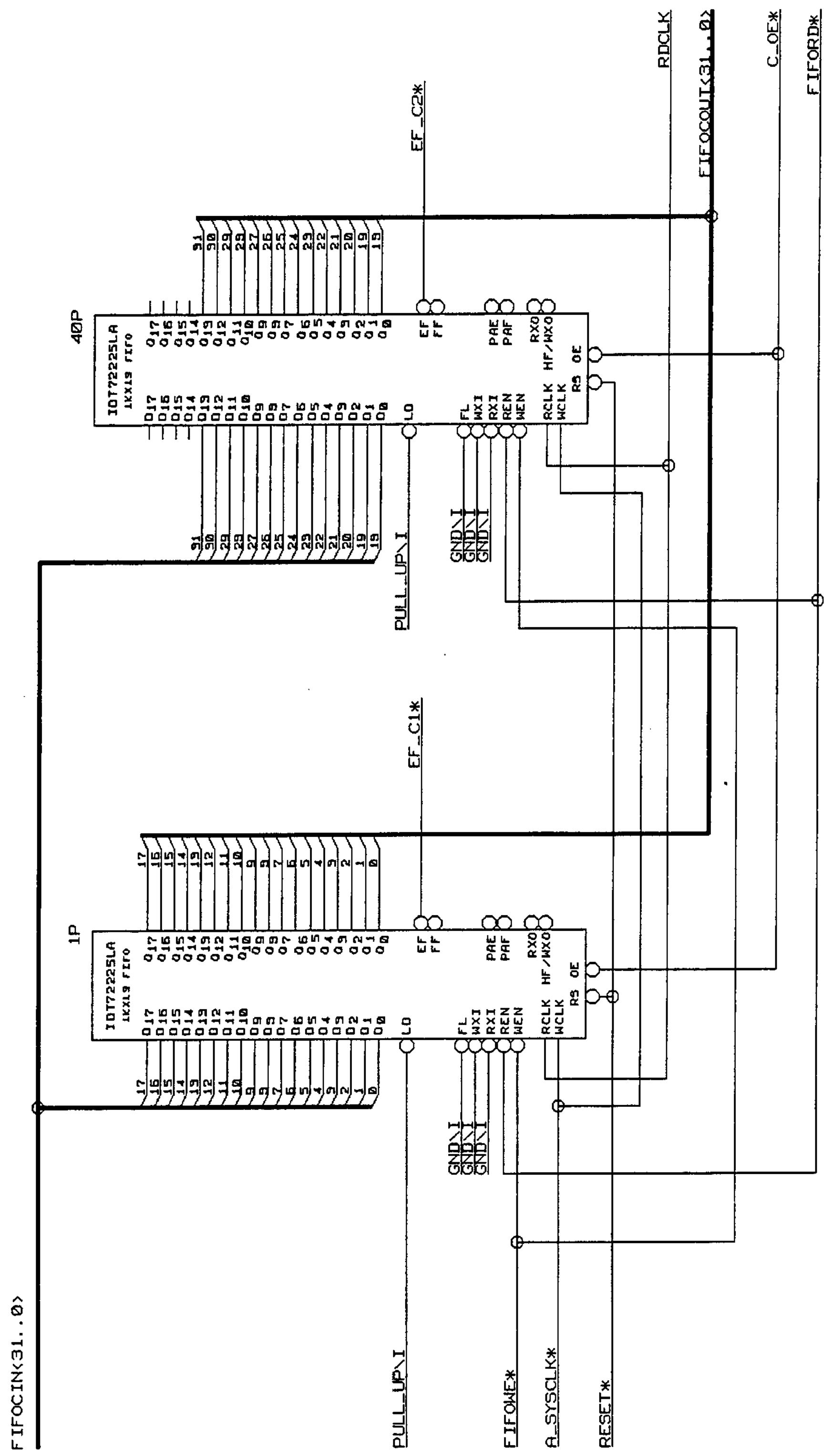


Figure 40. TMR Testbed Schematic (10 of 11).

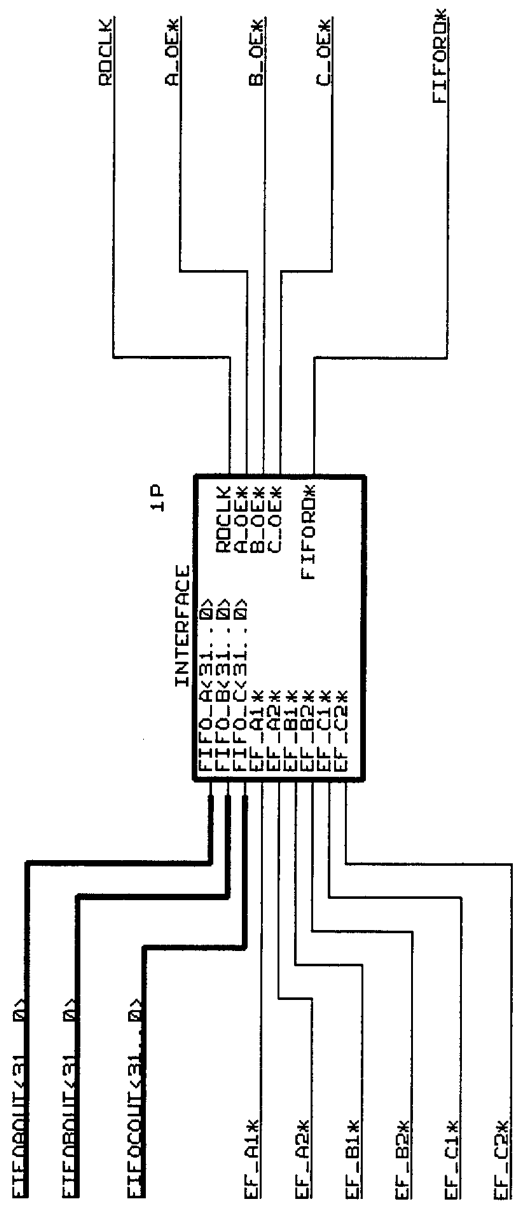


Figure 41. TMR Testbed Schematic (11 of 11).

APPENDIX B. CADENCE SUPPLIED MODULES

This appendix contains the TMR Testbed schematic modules, which were supplied in the Cadence Concept™ schematic libraries.

A. A74FCT373 TRANSPARENT LATCH

This part was used to build the address demultiplexer. The body diagram of the address demultiplexer and its schematic follow.

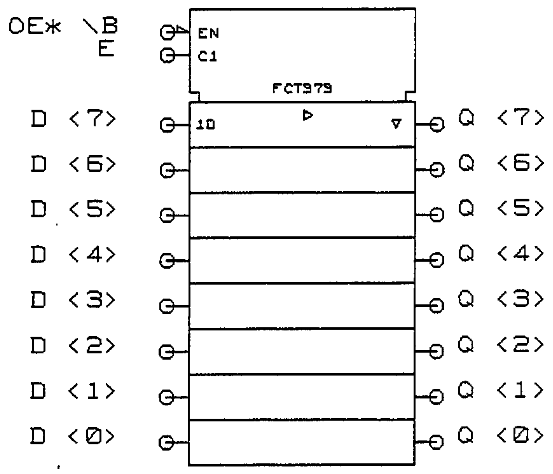


Figure 42. A74FCT373 Transparent Latch.

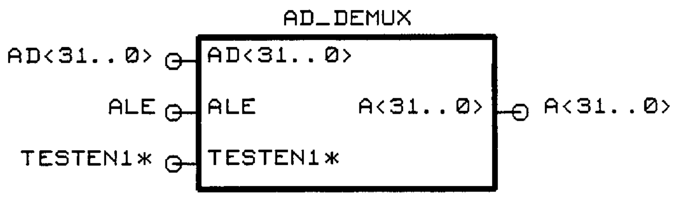


Figure 43. Address Demultiplexer.

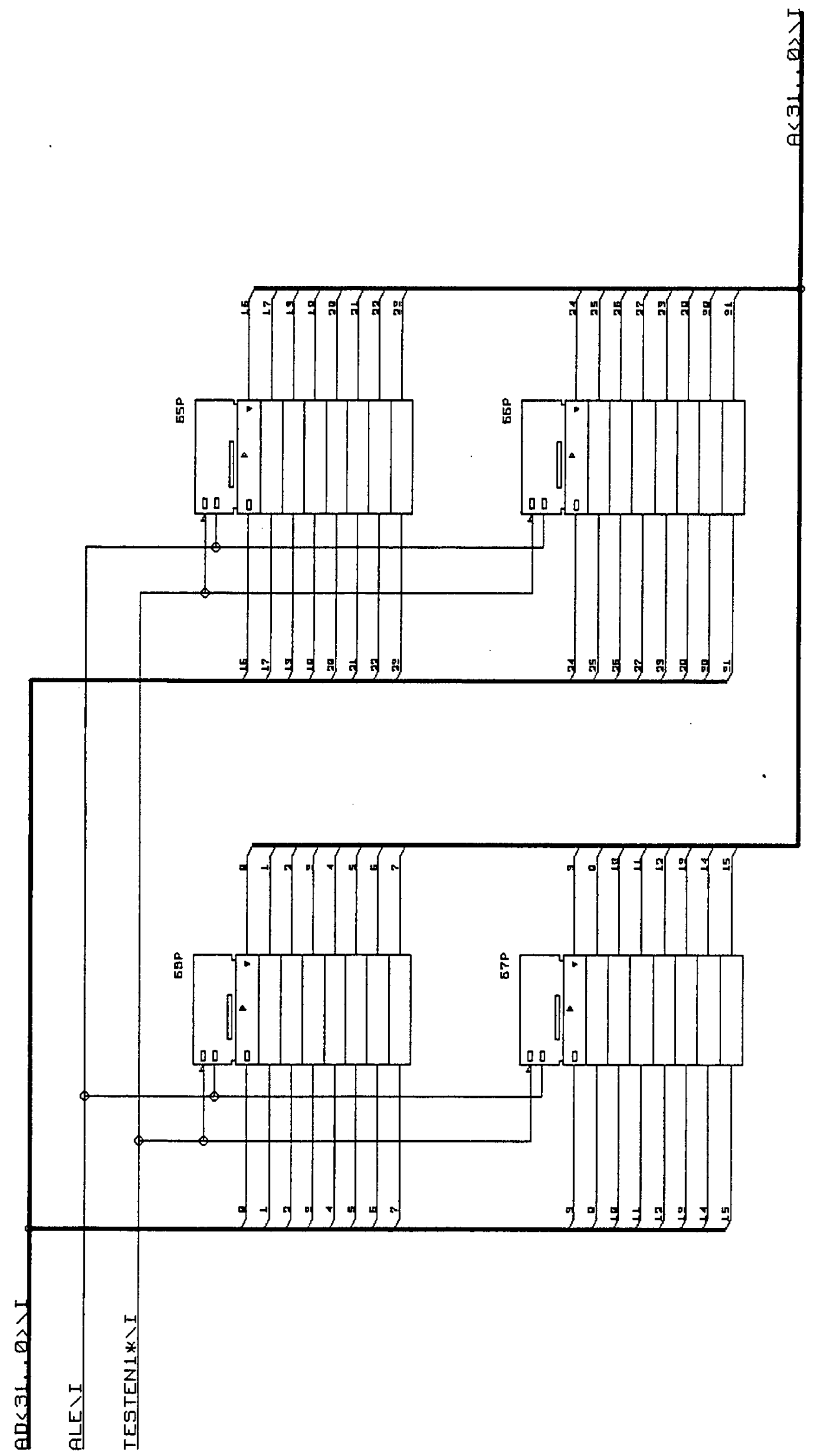


Figure 44. Address Demultiplexer Schematic.

B. IDT71256 32K X 8 SRAM

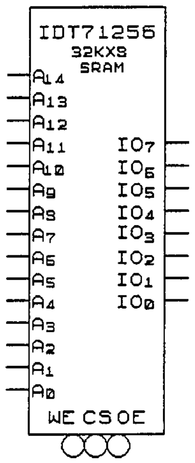


Figure 45. IDT71256 SRAM.

C. IDT72225LA 1K X 18 FIFO

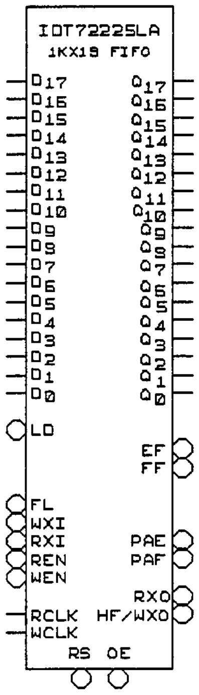


Figure 46. IDT72225LA FIFO.

APPENDIX C. USER DEFINED VERILOG® MODULES

This appendix contains the custom modules built using the Verilog® Hardware Description Language and the part body diagrams built using the Cadence Concept™ schematic tools.

A. IDT R3081 RISC MICROPROCESSOR BUS SIMULATOR

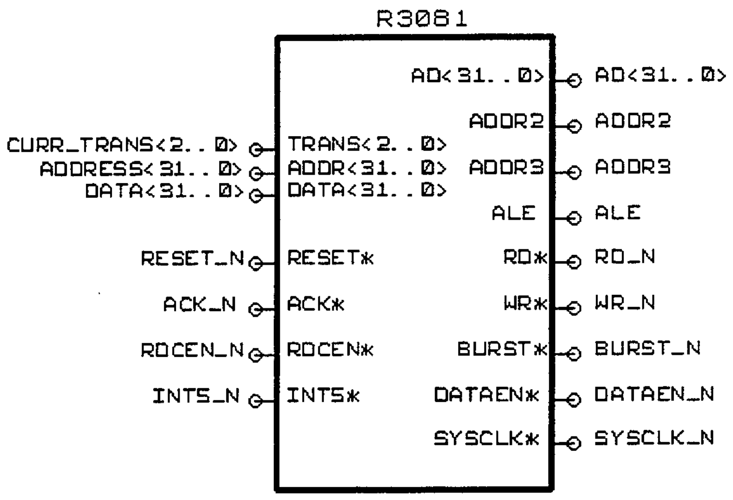


Figure 47. R3081 Microprocessor Bus Simulator.

```
//*******************************************************************
//* File:  r3081.v
//*
//* Description:  Verilog behavioral file for simulating the
//*    multiplexed address/data bus of a IDT RV3081.
//*
//* Reference:  (1) IDT79R3081 RISController with FPA Data Sheet
//*             (2) R3081 Family Hardware User's Guide
//*
//* Author:  John C. Payne, Jr.
//* Date:  10/24/98
//*******************************************************************

'timescale 1 ns /1 ps

'define NONE       0
'define READ_BYTE  1
'define READ_WORD  2
```

```
`define READ_BURST 3
`define WRITE_BYTE 4
`define WRITE_WORD 5
`define HIGH       1
`define LOW        0
`define TRUE       1
`define FALSE      0


//**********************************************************************
//* Module:  r3081
//*
//* Description:  Verilog behavioral module for simulating the
//*    multiplexed address/data bus and control lines of the IDT R3081.
//*    This module drives the R3081 block in the Cadence Concept
//*    schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*
//* Reference:  (1) IDT79R3081 RISController with FPA Data Sheet
//*             (2) R3081 Family Hardware User's Guide
//**********************************************************************
module r3081 (SYSCLK_N, RD_N, WR_N, AD, ADDR3, ADDR2, ALE,
              DATAEN_N, BURST_N, RDCEN_N, ACK_N, RESET_N,
              INT5_N, CURR_TRANS, ADDRESS, DATA);

   //* RV3081 @ 20MHz rise/fall time parameters (min,typ,max)
   parameter
      t7_min = 0,
      t7_typ = 2.5,   //* t7 = Valid from SYSCLK_N rising
      t7_max = 5,
      t8_min = 0,
      t8_typ = 2,     //* t8 = Asserted from SYSCLK_N rising
      t8_max = 4,
      t9_min = 0,
      t9_typ = 2,     //* t9 = Negated from SYSCLK_N falling
      t9_max = 4,
      t11_min = 0,
      t11_typ = 7.5,  //* t11 = Asserted from SYSCLK_N falling
      t11_max = 15,
      t14_min = 0,
      t14_typ = 0,    //* t14 = Driven from SYSCLK_N rising
      t14_max = 0,
      t15_min = 0,
      t15_typ = 3.5,  //* t15 = Negated from SYSCLK_N falling
      t15_max = 7,
      t16_min = 0,
      t16_typ = 3,    //* t16 = Valid from SYSCLK_N
      t16_max = 6,
      t18_min = 0,
      t18_typ = 5,    //* t18 = Tri-State from SYSCLK_N falling
      t18_max = 10,
      t19_min = 0,
      t19_typ = 6.5,  //* t19 = SYSCLK_N falling to data valid
      t19_max = 13;
```

```
//* Module input and output lines
output SYSCLK_N,
       RD_N,
       WR_N;
inout [31:0] AD;
output ADDR3,
       ADDR2,
       ALE,
       DATAEN_N,
       BURST_N;
input RDCEN_N,
      ACK_N,
      RESET_N,
      INT5_N;

//* These three inputs are not actual pins on an IDT R3081.  They
//* are used as interface pins to the bus simulator to command the
//* bus to initiate a read, burst read, or a write.
input [2:0] CURR_TRANS;
input [31:0] ADDRESS;
input [31:0] DATA;

reg SYSCLK_N;
wire RD_N, ADDR3, ADDR2, ALE, DATAEN_N, BURST_N;

//* Internal variables (line enables)
reg RD_N_enable;
reg WR_N_enable;
reg AD_enable;
reg ADDR3_enable;
reg ADDR2_enable;
reg ALE_enable;
reg DATAEN_N_enable;
reg BURST_N_enable;
reg [31:0] busValue;
reg startCycle;
reg bootCycle;
reg [31:0] saveAddress;
reg [31:0] saveData;

//* R3081 Multiplexed Address/Data Bus (32 bit)
busDriver     #(t14_min,t14_typ,t14_max,
                t18_min,t18_typ,t18_max,
                t18_min,t18_typ,t18_max)
             ADBus(AD, busValue, AD_enable);

//* R3081 Output Line RD_N Driver
activeLowLineDriver
   #(t15_min,t15_typ,t15_max,t7_min,t7_typ,t7_max)
      RDLine(RD_N, RD_N_enable);

//* R3081 Output Line WR_N Driver
activeLowLineDriver
   #(t15_min,t15_typ,t15_max,t7_min,t7_typ,t7_max)
      WRLine(WR_N, WR_N_enable);
```

```
//* R3081 Output Line ADDR3 Driver
activeHighLineDriver
   #(t16_min,t16_typ,t16_max,t16_min,t16_typ,t16_max)
      ADDR3Line(ADDR3, ADDR3_enable);

//* R3081 Output Line ADDR2 Driver
activeHighLineDriver
   #(t16_min,t16_typ,t16_max,t16_min,t16_typ,t16_max)
      ADDR2Line(ADDR2, ADDR2_enable);

//* R3081 Output Line ALE Driver
activeHighLineDriver
   #(t8_min,t8_typ,t8_max,t9_min,t9_typ,t9_max)
      ALELine(ALE, ALE_enable);

//* R3081 Output Line DATAEN_N Driver
activeLowLineDriver
   #(t15_min,t15_typ,t15_max,t11_min,t11_typ,t11_max)
      DATAENLine(DATAEN_N, DATAEN_N_enable);

//* R3081 Output Line BURST_N Driver
activeLowLineDriver
   #(t15_min,t15_typ,t15_max,t7_min,t7_typ,t7_max)
      BURSTLine(BURST_N, BURST_N_enable);

//* Initialize internal variables
initial
begin
   SYSCLK_N = 0;
   RD_N_enable = `LOW;
   WR_N_enable = `LOW;
   AD_enable = `LOW;
   ADDR3_enable = `LOW;
   ADDR2_enable = `LOW;
   ALE_enable = `LOW;
   DATAEN_N_enable = `LOW;
   BURST_N_enable = `LOW;
   busValue = 'bz;
   startCycle = `FALSE;
   saveAddress = 'bz;
   saveData = 'bz;
end

//* Control System Reference Clock
always
   #25 SYSCLK_N = ~ SYSCLK_N;

//* Watch for change in CURR_TRANS input.  If there is not a cycle
//* already started (startCycle = FALSE), then start a new cycle.
always @(CURR_TRANS)
   if (startCycle)
      startCycle = `FALSE;
   else if (CURR_TRANS == `NONE)
      startCycle = `FALSE;
   else
      startCycle = `TRUE;
```

```
//* At each positive edge of the system reference clock, if the
//* RESET_N input line is low, then set up system for initial burst
//* read from ROM at address 1FC00000
always @(posedge SYSCLK_N)
begin
   if (!RESET_N)
   begin
      busValue = 32'h1FC00000;
      AD_enable = 'HIGH;
      wait(RESET_N == 1);
      bootCycle = 'TRUE;
   end
end

//* Watch for negative edge of the interrupt line INT5_N.  If a
//* cycle is currently in progress, then it is a cycle that hasn't
//* finished because of an incorrect control input.  This means
//* that if, for example, this R3081 initiated a READ while the
//* other two R3081's initiated a WRITE, it will be stuck waiting
//* for signals from the memory controller which are associated
//* with a READ.  These signals will not come as expected because
//* the system completed a WRITE cycle based on the voted majority
//* from the other two R3081's.  After interrupting waiting
//* processor (if necessary), perform simulated, abbreviated
//* interrupt handler routine, beginning and ending the routine with
//* a WRITE to "dummy address" 1F800000
always @(negedge INT5_N)
begin
   if (!startCycle)  //* Then cycle is in progress
      case (CURR_TRANS[2:0])  //* Interrupt waiting cycle
         3'b001:
         begin  //* Interrupt a waiting READ_BYTE cycle
            disable readByte;
            @(negedge SYSCLK_N)
            begin
               RD_N_enable = 'LOW;
               DATAEN_N_enable = 'LOW;
               ADDR3_enable = 'LOW;
               ADDR2_enable = 'HIGH;
            @(posedge SYSCLK_N);
            end
         end

         3'b010:
         begin  //* Interrupt a waiting READ_WORD cycle
            disable readWord;
            @(negedge SYSCLK_N)
            begin
               RD_N_enable = 'LOW;
               DATAEN_N_enable = 'LOW;
               ADDR3_enable = 'LOW;
               ADDR2_enable = 'HIGH;
            @(posedge SYSCLK_N);
            end
         end
   3'b011:
   begin  //* Interrupt a waiting READ_BURST cycle
      disable readBurst;
      @(negedge SYSCLK_N)
      begin
         RD_N_enable = 'LOW;
         ADDR3_enable = 'LOW;
         ADDR2_enable = 'LOW;
         DATAEN_N_enable = 'LOW;
         BURST_N_enable = 'LOW;
      @(posedge SYSCLK_N);
      end
   end

   3'b100:
   begin  //* Interrupt a waiting WRITE_BYTE cycle
      disable writeByte;
      @(negedge SYSCLK_N)
         WR_N_enable = 'LOW;
      @(posedge SYSCLK_N)
      begin
         AD_enable = 'LOW;
         ADDR3_enable = 'LOW;
         ADDR2_enable = 'LOW;
      end
   end

   3'b101:
   begin  //* Interrupt a waiting WRITE_WORD cycle
      disable writeWord;
      @(negedge SYSCLK_N)
         WR_N_enable = 'LOW;
      @(posedge SYSCLK_N)
      begin
         AD_enable = 'LOW;
         ADDR3_enable = 'LOW;
         ADDR2_enable = 'LOW;
      end
   end

endcase

//* The saved address and data information from the last bus
//* cycle which caused the interrupt is used here simply to
//* show that differences between the three processors will
//* not cause a vote error interrupt to be generated due to
//* the WRITE to "dummy address" 1F800000.  The use of the
//* saved address and data information is not intended to
//* show what would actually be written during an interrupt
//* routine.
writeWord(32'h1F800000, 32'hFFFFFFFF);
if (saveAddress[31:0] >= 32'h00000000)
   writeWord(32'h00070000, saveAddress);
else
   writeWord(32'h00070000, 32'hA5A5A5A5);
if (saveData[31:0] >= 32'h00000000)
   writeWord(32'h00070004, saveData);
else
   writeWord(32'h00070004, 32'h78787878);
```

```
      readWord(32'h00070000);
      readWord(32'h00070004);
      writeWord(32'h1F800000, 32'hFFFFFFFF);

   end

//* Initiate appropriate bus cycles based on CURR_TRANS input, and
//* if startCyle is TRUE, or if a boot cycle is necessary.
//* See the simulated, abbreviated interrupt handler routine above
//* for how the saved address and data information is used.
always
begin
   if (startCycle && (CURR_TRANS == `READ_BYTE) && !bootCycle)
   begin
      saveAddress = ADDRESS;
      saveData = DATA;
      readByte(ADDRESS);
   end

   else if (startCycle && (CURR_TRANS == `READ_WORD) && !bootCycle)
   begin
      saveAddress = ADDRESS;
      saveData = DATA;
      readWord(ADDRESS);
   end

   else if ((startCycle && (CURR_TRANS == `READ_BURST))
            || bootCycle)
   begin
      saveAddress = ADDRESS;
      saveData = DATA;
      readBurst(ADDRESS);
   end

   else if (startCycle && (CURR_TRANS == `WRITE_BYTE) &&
            !bootCycle)
   begin
      saveAddress = ADDRESS;
      saveData = DATA;
      writeByte(ADDRESS, DATA);
   end

   else if (startCycle && (CURR_TRANS == `WRITE_WORD) &&
            !bootCycle)
   begin
      saveAddress = ADDRESS;
      saveData = DATA;
      writeWord(ADDRESS, DATA);
   end

   else
      @(posedge SYSCLK_N);
end
```

```
//*******************************************************************
//* task:  readByte
//*
//* Description:  Simulates the bus cycle for reading a byte from the
//*    given address by driving the A/D bus and associated control
//*    lines.  It waits on the RDCEN_N input from the memory
//*    controller to indicate the memory has placed valid data on the
//*    bus to read.
//*
//* Reference:  (1) IDT79R3081 RISController with FPA Data Sheet
//*             (2) R3081 Family Hardware User's Guide
//*******************************************************************
task readByte;
   input [31:0] address;

   begin:readByte
      @(posedge SYSCLK_N)
      begin
         startCycle = `FALSE;
         busValue[31:4] = address[31:4];

         //* Set BE[3:0] lines
         busValue[3] = !(address[1] && address[0]);
         busValue[2] = !(address[1] && !address[0]);
         busValue[1] = !(!address[1] && address[0]);
         busValue[0] = !(!address[1] && !address[0]);

         AD_enable = `HIGH;
         RD_N_enable = `HIGH;
         ADDR3_enable = address[3]; //* Set word address
         ADDR2_enable = address[2];
         ALE_enable = `HIGH;
      end

      @(negedge SYSCLK_N)
      begin
         AD_enable = `LOW;
         DATAEN_N_enable = `HIGH;
         ALE_enable = `LOW;
      end

      @(posedge RDCEN_N);

      @(negedge SYSCLK_N)
      begin
         RD_N_enable = `LOW;
         DATAEN_N_enable = `LOW;
         ADDR3_enable = `LOW;
         ADDR2_enable = `LOW;
      end
   end
endtask //* end task readByte
```

```
//*****************************************************************
//* task:  readWord
//*
//* Description:  Simulates the bus cycle for reading a word from the
//*    given address by driving the A/D bus and associated control
//*    lines.  It waits on the RDCEN_N input from the memory
//*    controller to indicate the memory has placed valid data on the
//*    bus to read.
//*
//* Reference:  (1) IDT79R3081 RISController with FPA Data Sheet
//*             (2) R3081 Family Hardware User's Guide
//*****************************************************************
task readWord;
   input [31:0] address;

   begin:readWord
      @(posedge SYSCLK_N)
      begin
         startCycle = `FALSE;
         busValue[31:4] = address[31:4];

         //* Set BE[3:0] lines
         busValue[3] = `LOW;
         busValue[2] = `LOW;
         busValue[1] = `LOW;
         busValue[0] = `LOW;

         AD_enable = `HIGH;
         RD_N_enable = `HIGH;
         ADDR3_enable = address[3];  //* Set word address
         ADDR2_enable = address[2];
         ALE_enable = `HIGH;
      end

      @(negedge SYSCLK_N)
      begin
         AD_enable = `LOW;
         DATAEN_N_enable = `HIGH;
         ALE_enable = `LOW;
      end

      @(posedge RDCEN_N);

      @(negedge SYSCLK_N)
      begin
         RD_N_enable = `LOW;
         DATAEN_N_enable = `LOW;
         ADDR3_enable = `LOW;
         ADDR2_enable = `LOW;
      end
   end
endtask //* end task readWord
```

```
//*******************************************************************
//* task:  readBurst
//*
//* Description:  Simulates the bus cycle for burst reading four
//*    contiguous words of memory starting at the given address
//*    by driving the A/D bus and associated control lines.
//*    It waits on the RDCEN_N four times input from the memory
//*    controller to indicate the memory has placed valid data on
//*    the bus to read.
//*
//* Reference:  (1) IDT79R3081 RISController with FPA Data Sheet
//*             (2) R3081 Family Hardware User's Guide
//*******************************************************************
task readBurst;
   input [31:0] address;

   begin:readBurst
      @(posedge SYSCLK_N)
      begin
         startCycle = `FALSE;
         if (!bootCycle)
            //* If it is a boot cycle, 1FC00000 will already
            //* be in busValue[31:0] for initial EPROM read

         begin
            busValue[31:4] = address[31:4];

            //* Set BE[3:0] lines
            busValue[3] = `LOW;
            busValue[2] = `LOW;
            busValue[1] = `LOW;
            busValue[0] = `LOW;
         end

         bootCycle = `FALSE;
         AD_enable = `HIGH;
         RD_N_enable = `HIGH;
         ADDR3_enable = `LOW;  //* Set word address of 1st word
         ADDR2_enable = `LOW;
         ALE_enable = `HIGH;
         BURST_N_enable = `HIGH;
      end

      @(negedge SYSCLK_N)
      begin
         AD_enable = `LOW;
         DATAEN_N_enable = `HIGH;
         ALE_enable = `LOW;
      end

      @(posedge RDCEN_N);  //* Wait for 1st word

      @(negedge SYSCLK_N)
      begin
         ADDR2_enable = `HIGH;  //* Set word address of 2nd word
      end

      @(posedge RDCEN_N);  //* Wait for 2nd word
      @(negedge SYSCLK_N)
      begin
         ADDR3_enable = `HIGH;  //* Set word address of 3rd word
         ADDR2_enable = `LOW;
      end

      @(posedge RDCEN_N);  //* Wait for 3rd word

      @(negedge SYSCLK_N)
      begin
         ADDR2_enable = `HIGH;  //* Set word address of 4th word
      end

      @(posedge RDCEN_N);  //* Wait for 4th word

      @(negedge SYSCLK_N)
      begin
         RD_N_enable = `LOW;
         ADDR3_enable = `LOW;
         ADDR2_enable = `LOW;
         DATAEN_N_enable = `LOW;
         BURST_N_enable = `LOW;
      end
   end
endtask //* end task readBurst


//*******************************************************************
//* task:  writeByte
//*
//* Description:  Simulates the bus cycle for writing a byte of the
//*    given data at the given address by driving the A/D bus and
//*    associated control lines.  It waits on the ACK_N input from
//*    the memory controller to indicate the data has been written.
//*
//* Reference:  (1) IDT79R3081 RISController with FPA Data Sheet
//*             (2) R3081 Family Hardware User's Guide
//*******************************************************************
task writeByte;
   input [31:0] address, data;

   begin:writeByte
      @(posedge SYSCLK_N)
      begin
         startCycle = `FALSE;
         busValue[31:4] = address[31:4];

         //* Set BE[3:0] lines
         busValue[3] = !(address[1] && address[0]);
         busValue[2] = !(address[1] && !address[0]);
         busValue[1] = !(!address[1] && address[0]);
         busValue[0] = !(!address[1] && !address[0]);

         AD_enable = `HIGH;
         WR_N_enable = `HIGH;
         ADDR3_enable = address[3];  //* Set word address
         ADDR2_enable = address[2];
         ALE_enable = `HIGH;
      end
      @(negedge SYSCLK_N)
      begin
         ALE_enable = `LOW;
         #(t19_min:t19_typ:t19_max)
            busValue = data;
      end

      @(posedge ACK_N);

      @(negedge SYSCLK_N)
      begin
         WR_N_enable = `LOW;
      end

      @(posedge SYSCLK_N)
      begin
         AD_enable = `LOW;
         ADDR3_enable = `LOW;
         ADDR2_enable = `LOW;
      end
   end
endtask //* end task writeByte


//*******************************************************************
//* task:  writeWord
//*
//* Description:  Simulates the bus cycle for writing a word of
//*    given data at the given address by driving the A/D bus and
//*    associated control lines.  It waits on the ACK_N input from
//*    the memory controller to indicate the data has been written.
//*
//* Reference:  (1) IDT79R3081 RISController with FPA Data Sheet
//*             (2) R3081 Family Hardware User's Guide
//*******************************************************************
task writeWord;
   input [31:0] address, data;

   begin:writeWord
      @(posedge SYSCLK_N)
      begin
         startCycle = `FALSE;
         busValue[31:4] = address[31:4];

         //* Set BE[3:0] lines
         busValue[3] = `LOW;
         busValue[2] = `LOW;
         busValue[1] = `LOW;
         busValue[0] = `LOW;

         AD_enable = `HIGH;
         WR_N_enable = `HIGH;
         ADDR3_enable = address[3];  //* Set word address
         ADDR2_enable = address[2];
         ALE_enable = `HIGH;
      end
```

```
         @(negedge SYSCLK_N)
         begin
            ALE_enable = 'LOW;
            #(t19_min:t19_typ:t19_max)
               busValue = data;
         end

         @(posedge ACK_N || !INT5_N);

         @(negedge SYSCLK_N)
         begin
            WR_N_enable = 'LOW;
         end

         @(posedge SYSCLK_N)
         begin
            AD_enable = 'LOW;
            ADDR3_enable = 'LOW;
            ADDR2_enable = 'LOW;
         end
      end
   endtask //* end task writeWord

endmodule //* end module r3081


//************************************************************************
//* Module:  busDriver
//*
//* Description:  Assigns valueToGo to address/data bus when driveEnable
//*    is HIGH, otherwise drives bus to high impedance.
//************************************************************************
module busDriver(busLine, valueToGo, driveEnable);
   parameter      //* Parameters may be overridden for each
                  //* instantiation of this module

      R_min = 0,  //* Minimum Rise Time
      R_typ = 2,  //* Typical Rise Time
      R_max = 4,  //* Maximum Rise Time
      F_min = 0,  //* Minimum Fall Time
      F_typ = 2,  //* Typical Fall Time
      F_max = 4,  //* Maximum Fall Time
      Z_min = 0,  //* Minimum Time to high impedance
      Z_typ = 2,  //* Typical Time to high impedance
      Z_max = 4;  //* Maximum Time to high impedance

      inout [31:0] busLine;
      input [31:0] valueToGo;
      input        driveEnable;

   assign #(R_min:R_typ:R_max,F_min:F_typ:F_max,Z_min:Z_typ:Z_max)
          busLine = (driveEnable)?valueToGo:'bz;

endmodule //* end module busDriver
```

```
//**********************************************************************
//* Module:  activeLowLineDriver
//*
//* Description:  Drives contLine LOW when driveEnable is HIGH,
//*    otherwise contLine remains HIGH.
//**********************************************************************
module activeLowLineDriver(contLine, driveEnable);
   parameter      //* Parameters may be overridden for each
                  //* instantiation of this module

      R_min = 0,  //* Minimum Rise Time
      R_typ = 2,  //* Typical Rise Time
      R_max = 4,  //* Maximum Rise Time
      F_min = 0,  //* Minimum Fall Time
      F_typ = 2,  //* Typical Fall Time
      F_max = 4;  //* Maximum Fall Time

   inout contLine;
   input driveEnable;

   assign #(R_min:R_typ:R_max,F_min:F_typ:F_max)
          contLine = (driveEnable)?0:1;

endmodule //* end module activeLowLineDriver


//**********************************************************************
//* Module:  activeLowLineDriver
//*
//* Description:  Drives contLine HIGH when driveEnable is HIGH,
//*    otherwise contLine remains LOW.
//**********************************************************************
module activeHighLineDriver(contLine, driveEnable);
   parameter      //* Parameters may be overridden for each
                  //* instantiation of this module

      R_min = 0,  //* Minimum Rise Time
      R_typ = 2,  //* Typical Rise Time
      R_max = 4,  //* Maximum Rise Time
      F_min = 0,  //* Minimum Fall Time
      F_typ = 2,  //* Typical Fall Time
      F_max = 4;  //* Maximum Fall Time

   inout contLine;
   input driveEnable;

   assign #(R_min:R_typ:R_max,F_min:F_typ:F_max)
          contLine = (driveEnable)?1:0;

endmodule //* end module activeHighLineDriver
```

B. 32-BIT VOTER/ERROR DETECTOR AND TRANSCEIVER

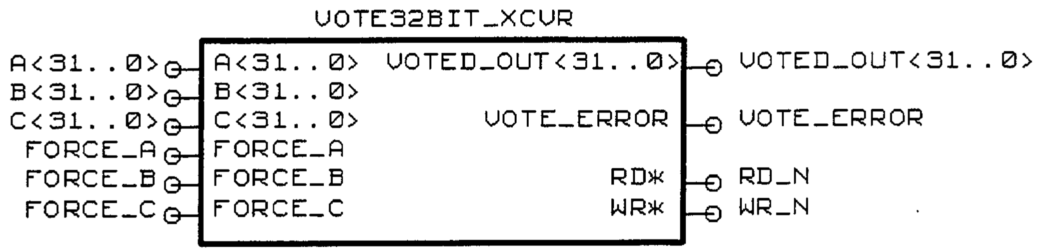


Figure 48. 32-Bit Voter/Error Detector and Transceiver.

```
//*********************************************************************
//* File:  vote32bit_xcvr.v
//*
//* Description:  Verilog file for a 32 bit majority voter/error
//*    detector and transceiver.
//*
//* Author:  John C. Payne, Jr.
//* Date:  10/31/98
//*********************************************************************

'timescale 1 ns /1 ps

//*********************************************************************
//* Module:  bidirsw
//*
//* Description:  Verilog behavioral module for a bidirectional switch
//*    with tristate.  If CONT_LINE is high, then the INOUT_LINE
//*    information drives the LINE_OUT line (LINE_OUT = INOUT_LINE);
//*    otherwise, the LINE_OUT line is in a high impedance state.  If
//*    CONT_LINE is low, then the LINE_IN information drives the
//*    INOUT_LINE (INOUT_LINE = LINE_IN); otherwise, the INOUT_LINE line
//*    is in a high impedance state.
//*********************************************************************
module bidirsw (LINE_IN, LINE_OUT, INOUT_LINE, CONT_LINE);

   input LINE_IN;
   output LINE_OUT;
   inout INOUT_LINE;
   input CONT_LINE;

   assign INOUT_LINE = (!CONT_LINE)?LINE_IN:'bz;
   assign LINE_OUT = (CONT_LINE)?INOUT_LINE:'bz;

endmodule  //* end module bidirsw
```

```
//*********************************************************************
//* Module:  votecell_xcvr
//*
//* Description:  Verilog structural module for a one bit voter/error
//*    detector and transceiver.  Votes 3 input bits to produce 1 output
//*    bit.  FORCE_A, FORCE_B, & FORCE_C inputs can be used to disable
//*    voting and force data on A, B, or C through to the output.
//*    Uses 4 bidirsw modules.
//*********************************************************************
module votecell_xcvr (A, B, C, FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
                       MAJ_OUT, MAJ_ERROR);

   inout A, B, C;
   input FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N;
   inout MAJ_OUT;
   output MAJ_ERROR;
   wire MAJORITY;

   tri IN_A, IN_B, IN_C, RD_IN;


   //* If RD_N is low, then RD_IN drives all three input/output lines
   //* A, B, & C; otherwise, A, B, & C drive IN_A, IN_B, & IN_C which
   //* are then voted.
   bidirsw
      sw_1(RD_IN, IN_A, A, RD_N),
      sw_2(RD_IN, IN_B, B, RD_N),
      sw_3(RD_IN, IN_C, C, RD_N);

   //* If WR_N is low, then MAJORITY drives the output line MAJ_OUT;
   //* otherwise, MAJ_OUT drives the RD_IN line.
   bidirsw
      sw_4(MAJORITY, RD_IN, MAJ_OUT, WR_N);

   not
      not_1 (NOT_IN_A, IN_A),
      not_2 (NOT_IN_B, IN_B),
      not_3 (NOT_IN_C, IN_C),
      not_4 (NOT_FORCE_A, FORCE_A),
      not_5 (NOT_FORCE_B, FORCE_B),
      not_6 (NOT_FORCE_C, FORCE_C);

   and
      and_1 (and_1_out, IN_A, FORCE_A),
      and_2 (and_2_out, IN_B, FORCE_B),
      and_3 (and_3_out, IN_C, FORCE_C),
      and_4 (and_4_out, IN_A, IN_B, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C),
      and_5 (and_5_out, IN_A, IN_C, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C),
      and_6 (and_6_out, IN_B, IN_C, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C);

   or #15
      or_1 (MAJORITY, and_1_out, and_2_out, and_3_out, and_4_out,
            and_5_out, and_6_out);
   and
      and_7 (and_7_out, NOT_IN_A, NOT_IN_B, IN_C, NOT_FORCE_A,
             NOT_FORCE_B, NOT_FORCE_C),
      and_8 (and_8_out, NOT_IN_A, IN_B, NOT_IN_C, NOT_FORCE_A,
             NOT_FORCE_B, NOT_FORCE_C),
      and_9 (and_9_out, NOT_IN_A, IN_B, IN_C, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C),
      and_10 (and_10_out, IN_A, NOT_IN_B, NOT_IN_C, NOT_FORCE_A,
              NOT_FORCE_B, NOT_FORCE_C),
      and_11 (and_11_out, IN_A, NOT_IN_B, IN_C, NOT_FORCE_A,
              NOT_FORCE_B, NOT_FORCE_C),
      and_12 (and_12_out, IN_A, IN_B, NOT_IN_C, NOT_FORCE_A,
              NOT_FORCE_B, NOT_FORCE_C);

   or #15
      or_2 (MAJ_ERROR, and_7_out, and_8_out, and_9_out, and_10_out,
            and_11_out, and_12_out);

endmodule  //* end module votecell_xcvr


//*******************************************************************
//* Module:  vote8bit_xcvr
//*
//* Description:  Verilog structural module for an 8 bit voter/error
//*    detector and transceiver.  Votes 24 input bits to produce 8
//*    output bits.  FORCE_A, FORCE_B, & FORCE_C inputs can be used to
//*    disable voting and force data on A[7:0], B[7:0], or C[7:0]
//*    through to the output.  Uses eight votecell_xcvr modules.
//*******************************************************************
module vote8bit_xcvr (A, B, C, FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
                      VOTED_OUT, VOTE_ERROR);

   inout [7:0] A, B, C;
   input FORCE_A, FORCE_B, FORCE_C;
   input RD_N, WR_N;
   inout [7:0] VOTED_OUT;
   output VOTE_ERROR;

   wire ERROR_0, ERROR_1, ERROR_2, ERROR_3, ERROR_4, ERROR_5, ERROR_6,
        ERROR_7;

   votecell_xcvr
      cell0 (A[0], B[0], C[0], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[0], ERROR_0),
      cell1 (A[1], B[1], C[1], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[1], ERROR_1),
      cell2 (A[2], B[2], C[2], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[2], ERROR_2),
      cell3 (A[3], B[3], C[3], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[3], ERROR_3),
      cell4 (A[4], B[4], C[4], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[4], ERROR_4),
      cell5 (A[5], B[5], C[5], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[5], ERROR_5),
      cell6 (A[6], B[6], C[6], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[6], ERROR_6),
      cell7 (A[7], B[7], C[7], FORCE_A, FORCE_B, FORCE_C, RD_N, WR_N,
             VOTED_OUT[7], ERROR_7);
   or #10
      or_1 (VOTE_ERROR, ERROR_0, ERROR_1, ERROR_2, ERROR_3, ERROR_4,
            ERROR_5, ERROR_6, ERROR_7);

endmodule  //* end module vote8bit_xcvr


//***********************************************************************
//* Module:  vote32bit_xcvr
//*
//* Description:  Verilog structural module for a 32 bit voter/error
//*    detector and transceiver.  Votes 96 input bits to produce 32
//*    output bits.  FORCE_A, FORCE_B, & FORCE_C inputs can be used
//*    to disable voting and force data on A[31:0], B[31:0], or C[31:0]
//*    through to the output.  Uses four vote8bit_xcvr modules.
//*    This module drives the VOTE32BIT_XCVR block in the Cadence
//*    Concept schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//***********************************************************************
module vote32bit_xcvr (A, B, C, FORCE_A, FORCE_B, FORCE_C,
                       RD_N, WR_N, VOTED_OUT, VOTE_ERROR);

   inout [31:0] A, B, C;
   input FORCE_A, FORCE_B, FORCE_C;
   input RD_N, WR_N;
   inout [31:0] VOTED_OUT;
   output VOTE_ERROR;

   wire ERROR_0, ERROR_1, ERROR_2, ERROR_3;

   vote8bit_xcvr
      voter0 (A[31:24], B[31:24], C[31:24], FORCE_A, FORCE_B, FORCE_C,
              RD_N, WR_N, VOTED_OUT[31:24], ERROR_0),
      voter1 (A[23:16], B[23:16], C[23:16], FORCE_A, FORCE_B, FORCE_C,
              RD_N, WR_N, VOTED_OUT[23:16], ERROR_1),
      voter2 (A[15:8], B[15:8], C[15:8], FORCE_A, FORCE_B, FORCE_C,
              RD_N, WR_N, VOTED_OUT[15:8], ERROR_2),
      voter3 (A[7:0], B[7:0], C[7:0], FORCE_A, FORCE_B, FORCE_C,
              RD_N, WR_N, VOTED_OUT[7:0], ERROR_3);

   or #10
      or_1 (VOTE_ERROR, ERROR_0, ERROR_1, ERROR_2, ERROR_3);

endmodule  //* end module vote32bit_xcvr
```

C. 8-BIT VOTER/ERROR DETECTOR

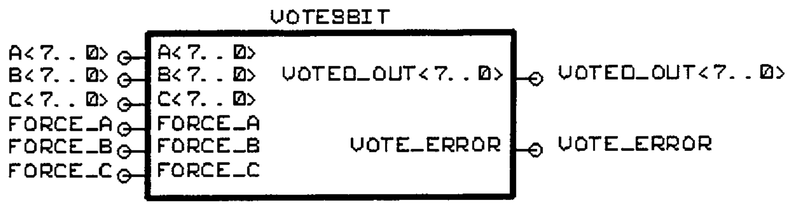


Figure 49. 8-Bit Voter/Error Detector.

```
//*******************************************************************
//* File:  vote8bit.v
//*
//* Description:  Verilog structural file for 8 bit majority voter and
//*               error detector using 8 votecell modules
//*
//* Author:  John C. Payne, Jr.
//* Date:  10/06/98
//*******************************************************************

'timescale 1 ns /1 ps

//*******************************************************************
//* Module:  votecell
//*
//* Description:  Verilog structural module for a one bit voter/error
//*    detector.  Votes 3 input bits to produce 1 output bit.  FORCE_A,
//*    FORCE_B, & FORCE_C inputs can be used to disable voting and
//*    force data on A, B, or C through to the output.
//*******************************************************************
module votecell (IN_A, FORCE_A, IN_B, FORCE_B, IN_C, FORCE_C, MAJ_OUT,
                 MAJ_ERROR);

   input IN_A, FORCE_A, IN_B, FORCE_B, IN_C, FORCE_C;
   output MAJ_OUT, MAJ_ERROR;

   not
      not_1 (NOT_IN_A, IN_A),
      not_2 (NOT_IN_B, IN_B),
      not_3 (NOT_IN_C, IN_C),
      not_4 (NOT_FORCE_A, FORCE_A),
      not_5 (NOT_FORCE_B, FORCE_B),
      not_6 (NOT_FORCE_C, FORCE_C);

   and
      and_1 (and_1_out, IN_A, FORCE_A),
      and_2 (and_2_out, IN_B, FORCE_B),
      and_3 (and_3_out, IN_C, FORCE_C),
      and_4 (and_4_out, IN_A, IN_B, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C),
      and_5 (and_5_out, IN_A, IN_C, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C),
      and_6 (and_6_out, IN_B, IN_C, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C);
   or  #15
      or_1 (MAJ_OUT, and_1_out, and_2_out, and_3_out, and_4_out,
            and_5_out, and_6_out);

   and
      and_7 (and_7_out, NOT_IN_A, NOT_IN_B, IN_C, NOT_FORCE_A,
             NOT_FORCE_B,  NOT_FORCE_C),
      and_8 (and_8_out, NOT_IN_A, IN_B, NOT_IN_C, NOT_FORCE_A,
             NOT_FORCE_B, NOT_FORCE_C),
      and_9 (and_9_out, NOT_IN_A, IN_B, IN_C, NOT_FORCE_A, NOT_FORCE_B,
             NOT_FORCE_C),
      and_10 (and_10_out, IN_A, NOT_IN_B, NOT_IN_C, NOT_FORCE_A,
              NOT_FORCE_B, NOT_FORCE_C),
      and_11 (and_11_out, IN_A, NOT_IN_B, IN_C, NOT_FORCE_A,
              NOT_FORCE_B, NOT_FORCE_C),
      and_12 (and_12_out, IN_A, IN_B, NOT_IN_C, NOT_FORCE_A,
              NOT_FORCE_B, NOT_FORCE_C);

   or #15
      or_2 (MAJ_ERROR, and_7_out, and_8_out, and_9_out, and_10_out,
            and_11_out, and_12_out);

endmodule  //* end module votecell


//*********************************************************************
//* Module:  vote8bit
//*
//* Description:  Verilog structural module for an 8 bit voter/error
//*    detector.  Votes 24 input bits to produce 8 output bits.
//*    FORCE_A, FORCE_B, & FORCE_C inputs can be used to disable voting
//*    and force data on A[7:0], B[7:0], or C[7:0] through to the
//*    output.  Uses eight votecell modules.  This module drives the
//*    VOTE8BIT block in the Cadence Concept schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*********************************************************************
module vote8bit (A, FORCE_A, B, FORCE_B, C, FORCE_C, VOTED_OUT,
                 VOTE_ERROR);

   input [7:0] A, B, C;
   input FORCE_A, FORCE_B, FORCE_C;
   output [7:0] VOTED_OUT;
   output VOTE_ERROR;

   wire ERROR_0, ERROR_1, ERROR_2, ERROR_3, ERROR_4, ERROR_5, ERROR_6,
        ERROR_7;

   votecell
      cell0 (A[0], FORCE_A, B[0], FORCE_B, C[0], FORCE_C,
             VOTED_OUT[0], ERROR_0),


      cell1 (A[1], FORCE_A, B[1], FORCE_B, C[1], FORCE_C,
             VOTED_OUT[1], ERROR_1),
      cell2 (A[2], FORCE_A, B[2], FORCE_B, C[2], FORCE_C,
             VOTED_OUT[2], ERROR_2),
      cell3 (A[3], FORCE_A, B[3], FORCE_B, C[3], FORCE_C,
             VOTED_OUT[3], ERROR_3),
      cell4 (A[4], FORCE_A, B[4], FORCE_B, C[4], FORCE_C,
             VOTED_OUT[4], ERROR_4),
      cell5 (A[5], FORCE_A, B[5], FORCE_B, C[5], FORCE_C,
             VOTED_OUT[5], ERROR_5),
      cell6 (A[6], FORCE_A, B[6], FORCE_B, C[6], FORCE_C,
             VOTED_OUT[6], ERROR_6),
      cell7 (A[7], FORCE_A, B[7], FORCE_B, C[7], FORCE_C,
             VOTED_OUT[7], ERROR_7);

   or #10
      or_1 (VOTE_ERROR, ERROR_0, ERROR_1, ERROR_2, ERROR_3, ERROR_4,
            ERROR_5, ERROR_6, ERROR_7);

endmodule  //* end module vote8bit
```

D. 32-BIT VOTER/ERROR DETECTOR

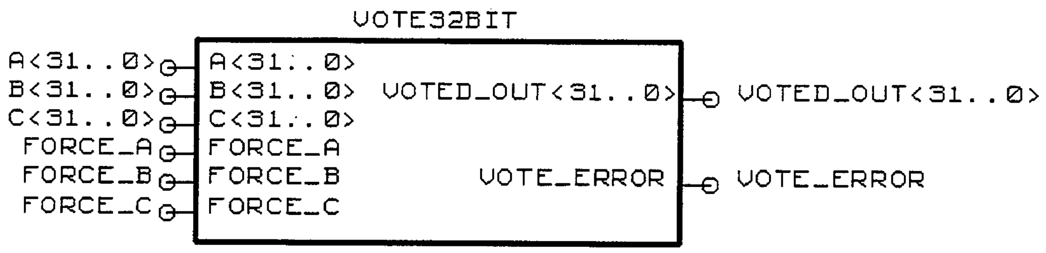


Figure 50. 32-Bit Voter/Error Detector.

```
//*********************************************************************
//* File:  vote32bit.v
//*
//* Description:  Verilog structural file for 32 bit majority voter and
//*               error detector using 4 voter_8bit modules
//*
//* Author:  John C. Payne, Jr.
//* Date:  10/06/98
//*********************************************************************

'timescale 1 ns /1 ps

//*********************************************************************
//* Module:  vote32bit
//*
//* Description:  Verilog structural module for a 32 bit voter/error
//*    detector.  Votes 96 input bits to produce 32 output bits.
//*    FORCE_A, FORCE_B, & FORCE_C inputs can be used to disable voting
//*    and force data on A[31:0], B[31:0], or C[31:0] through to the
//*    output.  Uses four vote8bit modules.  This module drives the
//*    VOTE32BIT block in the Cadence Concept schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*********************************************************************
module vote32bit (A, FORCE_A, B, FORCE_B, C, FORCE_C, VOTED_OUT,
                  VOTE_ERROR);

   input [31:0] A, B, C;
   input FORCE_A, FORCE_B, FORCE_C;
   output [31:0] VOTED_OUT;
   output VOTE_ERROR;

   wire ERROR_0, ERROR_1, ERROR_2, ERROR_3;
```

```
 vote8bit
      voter0 (A[31:24], FORCE_A, B[31:24], FORCE_B, C[31:24], FORCE_C,
             VOTED_OUT[31:24], ERROR_0),
      voter1 (A[23:16], FORCE_A, B[23:16], FORCE_B, C[23:16], FORCE_C,
             VOTED_OUT[23:16], ERROR_1),
      voter2 (A[15:8], FORCE_A, B[15:8], FORCE_B, C[15:8], FORCE_C,
             VOTED_OUT[15:8], ERROR_2),
      voter3 (A[7:0], FORCE_A, B[7:0], FORCE_B, C[7:0], FORCE_C,
             VOTED_OUT[7:0], ERROR_3);

   or #10
      or_1 (VOTE_ERROR, ERROR_0, ERROR_1, ERROR_2, ERROR_3);

endmodule  //* end module vote32bit
```

E. MEMORY/ADDRESS DECODER

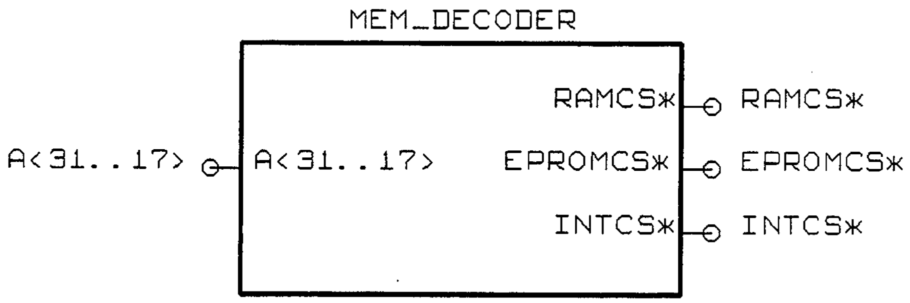


Figure 51. Memory/Address Decoder.

```
//*********************************************************************
//* File:  mem_decoder.v
//*
//* Description:  Verilog structural file for memory decoder to
//*               generate various chip selects.
//*
//* Author:  John C. Payne, Jr.
//* Date:  10/06/98
//*********************************************************************

`timescale 1 ns / 1 ps

//*********************************************************************
//* Module:  mem_decoder
//*
//* Description:  Verilog behavioral module for a memory decoder.  Uses
//*    input A[31:17] to generate three active low chip select outputs.
//*    This module drives the MEM_DECODER block in the Cadence Concept
//*    schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*********************************************************************
module mem_decoder (A, RAMCS_N, EPROMCS_N, INTCS_N);

   input [31:17] A;
   output RAMCS_N, EPROMCS_N, INTCS_N;
   wire RAMCS_N, EPROMCS_N, INTCS_N;

//*   RAM   = 00000000 to 0007FFFF
//*   EPROM = 1FC00000 to 1FC0xxxx
//*   INT   = 1F800000 "Dummy Address to Disable Vote Error Interrupts"
```

```
   assign #45
      RAMCS_N =    (!A[31] && !A[30] && !A[29] && !A[28] && //* 0
                    !A[27] && !A[26] && !A[25] && !A[24] && //* 0
                    !A[23] && !A[22] && !A[21] && !A[20] && //* 0
                    !A[19])?0:1;                            //* 7

   assign #45
      EPROMCS_N = (!A[31] && !A[30] && !A[29] &&  A[28] && //* 1
                    A[27] &&  A[26] &&  A[25] &&  A[24] && //* F
                    A[23] &&  A[22] && !A[21] && !A[20] && //* C
                   !A[19] && !A[18] && !A[17])?0:1;         //* 0

   assign #45
      INTCS_N =    (!A[31] && !A[30] && !A[29] &&  A[28] && //* 1
                     A[27] &&  A[26] &&  A[25] &&  A[24] && //* F
                     A[23] && !A[22] && !A[21] && !A[20] && //* 8
                    !A[19] && !A[18] && !A[17])?0:1;         //* 0

endmodule  //* end module mem_decoder
```

F. MEMORY/ERROR CONTROLLER

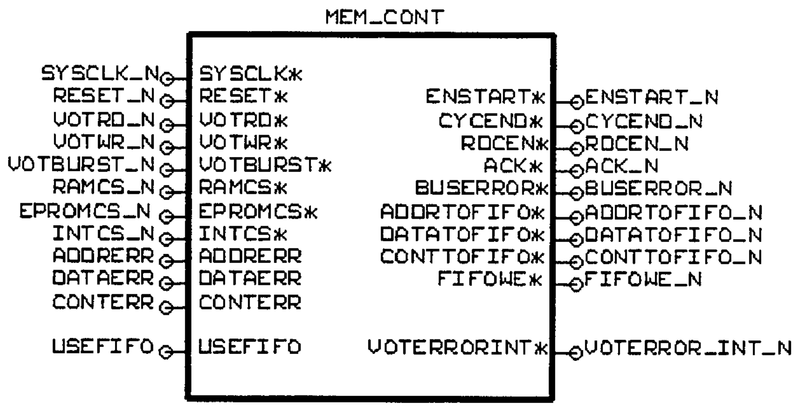


Figure 52. Memory/Error Controller.

```
//************************************************************************
//* File:  mem_cont.v
//*
//* Description:  Verilog behavioral file for memory/error controller
//*               to control timing cycles of various bus transactions.
//*
//* Reference:  (1) IDT RISC Microprocessor Application Guide,
//*                 Application Note AN-86, IDT79R3051 System Design
//*                 Example
//*
//* Author:  John C. Payne, Jr.
//* Date:  11/3/98
//************************************************************************

`timescale 1 ns /1 ps


//************************************************************************
//* Module:  mem_cont
//*
//* Description:  Verilog behavioral module for the memory/error
//*    controller.  Produces READ, WRITE, and BUS ERROR acknowledge
//*    controls (RDCEN_N, ACK_N, BUSERROR_N) based on a 5 bit counter
//*    and cycle end stall cycle (wait state) equations.
//*    Also produces an interrupt if there is a vote error detected on
//*    the ADDRERR, CONTERR, or DATAERR inputs.  The ADDRERR, CONTERR,
//*    and DATAERR inputs are saved at specified values of the counter,
//*    and an error interrupt is generated only at the end of the
//*    current cycle, so that the current cycle is allowed to finish.
//*    If INTCS_N goes low during a dummy write to that address, this
//*    signals the beginning of the interrupt handler routine and
//*    vote error interrupts are disabled until INTCS_N goes low again,
//*    which signals the end of the interrupt handler routine.
//*    This module also controls the three lines ADDRTOFIFO_N,
//*    CONTTOFIFO_N, and DATATOFIFO_N which send the appropriate
//*    information to the dedicated FIFOs.  These three lines are
//*    active low enable lines which allow, through the use of 32-bit
//*    tri-state buffers, the ADDRESS, CONTROL, and DATA information
//*    from the processor to be multiplexed onto a single 32-bit bus
//*    which is the input bus for each dedicated FIFO.  The FIFOWE_N
//*    line signals a write to the FIFOs at the appropriate time within
//*    a bus cycle based on the 5-bit counter.
//*    This module drives the MEM_CONT block in the Cadence Concept
//*    schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*
//* Reference:  (1) IDT RISC Microprocessor Application Guide,
//*                 Application Note AN-86, IDT79R3051 System Design
//*                 Example
//*
//*******************************************************************
module mem_cont (SYSCLK_N, RESET_N, VOTRD_N, VOTWR_N, VOTBURST_N,
                 RAMCS_N, EPROMCS_N, INTCS_N, USEFIFO, DATAERR,
                 ADDRERR, CONTERR, ENSTART_N, CYCEND_N,
                 RDCEN_N, ACK_N, BUSERROR_N, ADDRTOFIFO_N,
                 DATATOFIFO_N, CONTTOFIFO_N, FIFOWE_N,
                 VOTERROR_INT_N);

   input SYSCLK_N,      //* System clock from R3081
         RESET_N,       //* Reset from MEMEN module
         VOTRD_N,       //* Voted read from R3081
         VOTWR_N,       //* Voted write from R3081
         VOTBURST_N,    //* Voted burst from R3081
         RAMCS_N,       //* RAM chip select from memory decoder
         EPROMCS_N,     //* EPROM chip select from memory decoder
         INTCS_N,       //* INT chip select from memory decoder
         USEFIFO,       //* Set High (pull up) to Write to FIFOs
         DATAERR,       //* Data Vote Error from 32-bit Data Voter
         ADDRERR,       //* Address Vote Error from 32-bit Address Voter
         CONTERR;       //* Control Vote Error from 8-bit Control Voter

   output ENSTART_N,       //* Read/write output enable start
          CYCEND_N,        //* Cyle end (composite ACK)
          RDCEN_N,         //* R3081 read buffer clock enable
          ACK_N,           //* R3081 acknowledge
          BUSERROR_N,      //* R3081 bus error
          ADDRTOFIFO_N,    //* Address To FIFO to Address Buffers
          DATATOFIFO_N,    //* Data To FIFO to Data Buffers
          CONTTOFIFO_N,    //* Control To FIFO to Control Buffers
          FIFOWE_N,        //* FIFO Write Enable
          VOTERROR_INT_N;  //* Interrupt Sent to R3081

   wire ENSTART_N, CYCEND_N, RDCEN_N, ACK_N, BUSERROR_N,
        ADDRTOFIFO_N, DATATOFIFO_N, CONTTOFIFO_N,
        FIFOWE_N, VOTERROR_INT_N;

   reg [4:0] counter;
   reg voteErrorIntEn;
   wire voteError;
   reg saveError1, saveError2, saveError3, saveError4;
   reg voteErrorIntValueToGo;
```

```
//* At the positive edge of the reset input line, RESET_N, ensure
//* vote error interrupts are enabled, the interrupt line is HIGH,
//* and the saved error flags are initialized to indicated no error
//* has been detected.
always
   @(posedge RESET_N)
      begin
         voteErrorIntEn = 1;
         voteErrorIntValueToGo = 1;
         saveError1 = 0;
         saveError2 = 0;
         saveError3 = 0;
         saveError4 = 0;
      end

//* At each positive edge of the system reference clock generated
//* by the R3081, reset the counter if RESET_N or CYCEND_N goes low.
//* Increment the counter if VOTRD_N or VOTWR_N is low.  Save the
//* error flag at the four different counter values, so that the
//* cycle is allowed to finish.  The use of four different saved
//* values allows a single READ or WRITE to finish as well as a
//* BURST READ to finish.  If the current transaction is a BURST
//* READ, then an ADDRERR, CONTERR, or DATAERR is sampled four times.
always
   @(posedge SYSCLK_N)
   begin
      if (!RESET_N || !CYCEND_N)
         counter = 0;
      else if (!VOTRD_N || !VOTWR_N)
         counter = counter + 1;

      if (RESET_N && CYCEND_N && (counter == 5'h05))
         saveError1 = voteError;
      else if (RESET_N && CYCEND_N && (counter == 5'h09))
         saveError2 = voteError;
      else if (RESET_N && CYCEND_N && (counter == 5'h0B))
         saveError3 = voteError;
      else if (RESET_N && CYCEND_N && (counter == 5'h17))
         saveError4 = voteError;

      //* If at the end of a cycle, and one of the saved errors
      //* indicates an error occurred, then generate an interrupt
      //* only if vote error interrupts are currently enabled.
      if (RESET_N && !CYCEND_N && voteErrorIntEn &&
            (saveError1 || saveError2 || saveError3 || saveError4))
         voteErrorIntValueToGo = 0;
   end

//* Watch for negative edge of INTCS_N, and disable/reenable vote
//* error interrupts.
always
   @(negedge INTCS_N)
   begin
      voteErrorIntEn = ~voteErrorIntEn;
      voteErrorIntValueToGo = 1;
      saveError1 = 0;
      saveError2 = 0;
      saveError3 = 0;
      saveError4 = 0;
   end

//* Update internal voteError flag
assign #30 voteError = (ADDRERR || DATAERR || CONTERR)?1:0;

//* Update VOTERROR_INT_N output line
assign #30 VOTERROR_INT_N = voteErrorIntValueToGo;

//* Update ENSTART_N output line
assign #30 ENSTART_N = (RESET_N && (counter >= 1) && CYCEND_N)?0:1;

//* Update CYCEND_N output line
assign #30 CYCEND_N =
   (RESET_N && CYCEND_N && (
      (!RAMCS_N && (counter == 5'h05) && !VOTRD_N && VOTBURST_N)
   || (!RAMCS_N && (counter == 5'h17) && !VOTRD_N && !VOTBURST_N)
   || (!RAMCS_N && (counter == 5'h06) && !VOTWR_N)
   || (!EPROMCS_N && (counter == 5'h05) && !VOTRD_N && VOTBURST_N)
   || (!EPROMCS_N && (counter == 5'h17) && !VOTRD_N && !VOTBURST_N)
   || (!INTCS_N && (counter == 5'h06) && !VOTWR_N)
   || (counter == 8'h1F)
   ))?0:1;

//* Update RDCEN_N output line
assign #30 RDCEN_N =
   (RESET_N && CYCEND_N && (
      (!RAMCS_N && !VOTRD_N &&
         (
                         (counter == 5'h03)
      || (!VOTBURST_N && (counter == 5'h09))
      || (!VOTBURST_N && (counter == 5'h0F))
      || (!VOTBURST_N && (counter == 5'h15))
         )
      )
   || (!EPROMCS_N && !VOTRD_N &&
         (
                         (counter == 5'h03)
      || (!VOTBURST_N && (counter == 5'h09))
      || (!VOTBURST_N && (counter == 5'h0F))
      || (!VOTBURST_N && (counter == 5'h15))
         )
      )
   ))?0:1;

//* Update ACK_N output line
assign #30 ACK_N = (RESET_N && CYCEND_N &&
                   (
                       (!RAMCS_N && !VOTWR_N &&
                          (counter == 5'h06)
                       )
                    || (!RAMCS_N && !VOTRD_N &&
                          (counter == 5'h03)
                       )
                    || (!EPROMCS_N && !VOTRD_N &&
                          (counter == 5'h03)
                       )
                    || (!INTCS_N && !VOTWR_N &&
                           (counter == 5'h06)
                       )
                   ))?0:1;

//* Update BUSERROR_N output line
assign #30 BUSERROR_N =
   (RESET_N && CYCEND_N && (counter == 5'h1F))?0:1;

//* Update ADDRTOFIFO_N output line
assign #30 ADDRTOFIFO_N =
   (RESET_N && CYCEND_N && USEFIFO &&
      (
                                 (counter == 5'h01)
         || (!EPROMCS_N && !VOTRD_N &&
               (
                  (!VOTBURST_N && (counter == 5'h07))
               || (!VOTBURST_N && (counter == 5'h0D))
               || (!VOTBURST_N && (counter == 5'h13))
               )
            )
         || (!RAMCS_N && !VOTRD_N &&
               (
                  (!VOTBURST_N && (counter == 5'h07))
               || (!VOTBURST_N && (counter == 5'h0D))
               || (!VOTBURST_N && (counter == 5'h13))
               )
            )
      )
   )?0:1;

//* Update CONTTOFIFO_N output line
assign #30 CONTTOFIFO_N =
   (RESET_N && CYCEND_N && USEFIFO &&
      (
                                 (counter == 5'h03)
         || (!EPROMCS_N && !VOTRD_N &&
               (
                  (!VOTBURST_N && (counter == 5'h09))
               || (!VOTBURST_N && (counter == 5'h0F))
               || (!VOTBURST_N && (counter == 5'h15))
               )
            )
         || (!RAMCS_N && !VOTRD_N &&
               (
                  (!VOTBURST_N && (counter == 5'h09))
               || (!VOTBURST_N && (counter == 5'h0F))
               || (!VOTBURST_N && (counter == 5'h15))
               )
            )
      )
   )?0:1;
```

```
    //* Update DATATOFIFO_N output line
    assign #30 DATATOFIFO_N =
       (RESET_N && CYCEND_N && USEFIFO &&
          (
                                            (counter == 5'h05)
             || (!EPROMCS_N && !VOTRD_N &&
                   (
                      (!VOTBURST_N && (counter == 5'h0B))
                   || (!VOTBURST_N && (counter == 5'h11))
                   || (!VOTBURST_N && (counter == 5'h17))
                   )
                )
             || (!RAMCS_N && !VOTRD_N &&
                   (
                      (!VOTBURST_N && (counter == 5'h0B))
                   || (!VOTBURST_N && (counter == 5'h11))
                   || (!VOTBURST_N && (counter == 5'h17))
                   )
                )
          )
       )?0:1;

    //* Update FIFOWE_N output line
    assign #30 FIFOWE_N =
       (RESET_N && CYCEND_N && USEFIFO &&
          (
                (counter == 5'h01)
             || (counter == 5'h03)
             || (counter == 5'h05)
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h07))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h09))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h0B))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h0D))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h0F))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h11))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h13))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h15))
             || (!VOTBURST_N && !VOTRD_N && (counter == 5'h17))
          )
       )?0:1;

endmodule //* end module mem_cont
```

G. MEMORY READ/WRITE ENABLE CONTROLLER

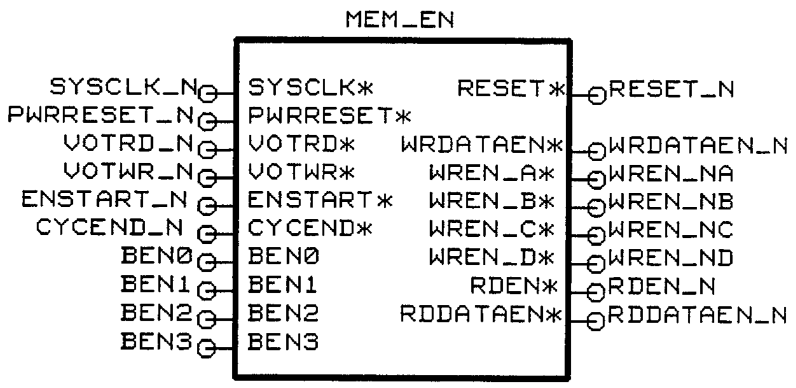


Figure 53. Memory Read/Write Enable Controller.

```
//**********************************************************************
//* File:  mem_en.v
//*
//* Description:  Verilog behavioral file for generating memory read
//*               and write enable signals.
//*
//* Reference:  (1) IDT RISC Microprocessor Application Guide,
//*                 Application Note AN-86, IDT79R3051 System Design
//*                 Example
//*
//* Author:  John C. Payne, Jr.
//* Date:  11/1/98
//**********************************************************************

`timescale 1 ns /1 ps


//**********************************************************************
//* Module:  mem_cont
//*
//* Description:  Verilog behavioral module for generating the read
//*    and write enables for the memory controls.
//*    This module drives the MEM_EN block in the Cadence Concept
//*    schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*
//* Reference:  (1) IDT RISC Microprocessor Application Guide,
//*                 Application Note AN-86, IDT79R3051 System Design
//*                 Example
//*
//**********************************************************************
```

```
module mem_en (SYSCLK_N, PWRRESET_N, VOTRD_N, VOTWR_N, ENSTART_N,
               CYCEND_N, BEN0, BEN1, BEN2, BEN3, RESET_N, WREN_N,
               WRDATAEN_N, WREN_NA, WREN_NB, WREN_NC, WREN_ND, RDEN_N,
               RDDATAEN_N);

   input SYSCLK_N,    //* System clock from R3081
         PWRRESET_N,  //* Power (Global) reset
         VOTRD_N,     //* Voted read from R3081
         VOTWR_N,     //* Voted write from R3081
         ENSTART_N,   //* Enable start from memory controller
         CYCEND_N,    //* Cycle end from memory controller
         BEN0,        //* Byte 0 enable (active low) from R3081 (ADDR[0])
         BEN1,        //* Byte 1 enable (active low) from R3081 (ADDR[1])
         BEN2,        //* Byte 2 enable (active low) from R3081 (ADDR[2])
         BEN3;        //* Byte 3 enable (active low) from R3081 (ADDR[3])

   output RESET_N,      //* Synchronzied reset line to rest of board
          WREN_N,       //* Not used
          WRDATAEN_N,   //* Write data xcvr enable
          WREN_NA,      //* Write enable for byte 0
          WREN_NB,      //* Write enable for byte 1
          WREN_NC,      //* Write enable for byte 2
          WREN_ND,      //* Write enable for byte 3
          RDEN_N,       //* Read output enable (for words)
          RDDATAEN_N;   //* Read data xcvr enable

   wire RESET_N, WREN_N, WRDATAEN_N, WREN_NA, WREN_NB, WREN_NC, WREN_ND,
        RDEN_N, RDDATAEN_N;


   assign #30 WREN_NA =
      !(RESET_N &&
          (!VOTWR_N && !BEN0 && !ENSTART_N && CYCEND_N)
       );

   assign #30 WREN_NB =
      !(RESET_N &&
          (!VOTWR_N && !BEN1 && !ENSTART_N && CYCEND_N)
       );

   assign #30 WREN_NC =
      !(RESET_N &&
          (!VOTWR_N && !BEN2 && !ENSTART_N && CYCEND_N)
       );

   assign #30 WREN_ND =
      !(RESET_N &&
          (!VOTWR_N && !BEN3 && !ENSTART_N && CYCEND_N)
       );

   assign #30 WREN_N =
      !(RESET_N &&
          ((!VOTWR_N && CYCEND_N) || (!WREN_N && !CYCEND_N))
       );
   assign #30 WRDATAEN_N =
      !(RESET_N &&
           ((!VOTWR_N && !ENSTART_N) ||
            (!WRDATAEN_N && (!ENSTART_N || !CYCEND_N))
           )
       );

   assign #30 RDEN_N =
      !(RESET_N &&
           (!VOTRD_N && !ENSTART_N && CYCEND_N)
       );

   assign #30 RDDATAEN_N =
      !(RESET_N &&
           (!VOTRD_N && !ENSTART_N && CYCEND_N)
       );

   assign #30 RESET_N = !(!PWRRESET_N);

endmodule //* end module mem_en
```

H. 16-BIT NON-INVERTING TRI-STATE BUFFER

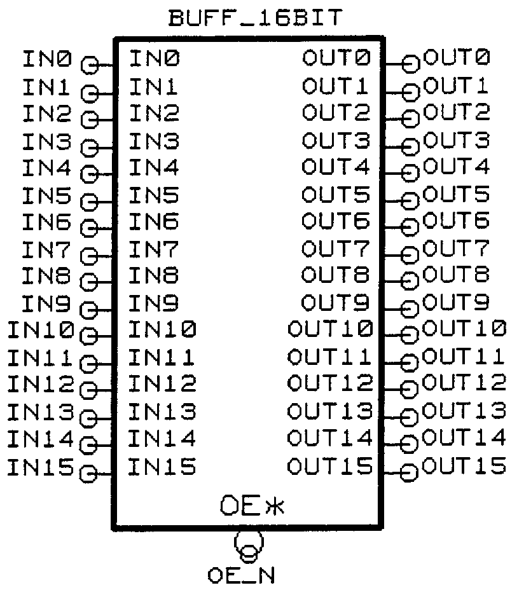


Figure 54. 16-Bit Non-Inverting Tri-State Buffer.

```
//***********************************************************************
//* File:  buff_16bit.v
//*
//* Description:  Verilog structural file for 16 bit tri-state
//*               non-inverting buffer.
//*
//* Author:  John C. Payne, Jr.
//* Date:  11/16/98
//***********************************************************************

`timescale 1 ns /1 ps


//***********************************************************************
//* Module:  interface
//*
//* Description:  Verilog structural module for simulating a 16-bit
//*    tri-state non-inverting buffer.
//*    This module drives the BUFF_16BIT block in the Cadence Concept
//*    schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//***********************************************************************
module buff_16bit (IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7,
                   IN8, IN9, IN10, IN11, IN12, IN13, IN14, IN15,
                   OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7,
                   OUT8, OUT9, OUT10, OUT11, OUT12, OUT13, OUT14, OUT15,
                   OE_N);
```

```
    input IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7,
          IN8, IN9, IN10, IN11, IN12, IN13, IN14, IN15;
    output OUT0, OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7,
           OUT8, OUT9, OUT10, OUT11, OUT12, OUT13, OUT14, OUT15;
    input OE_N;

    bufif0 #(0:15:30, 0:15:30, 0:15:30)
       buff_0 (OUT0, IN0, OE_N),
       buff_1 (OUT1, IN1, OE_N),
       buff_2 (OUT2, IN2, OE_N),
       buff_3 (OUT3, IN3, OE_N),
       buff_4 (OUT4, IN4, OE_N),
       buff_5 (OUT5, IN5, OE_N),
       buff_6 (OUT6, IN6, OE_N),
       buff_7 (OUT7, IN7, OE_N),
       buff_8 (OUT8, IN8, OE_N),
       buff_9 (OUT9, IN9, OE_N),
       buff_10 (OUT10, IN10, OE_N),
       buff_11 (OUT11, IN11, OE_N),
       buff_12 (OUT12, IN12, OE_N),
       buff_13 (OUT13, IN13, OE_N),
       buff_14 (OUT14, IN14, OE_N),
       buff_15 (OUT15, IN15, OE_N);

endmodule //* end module buff_16bit
```

I. EPROM

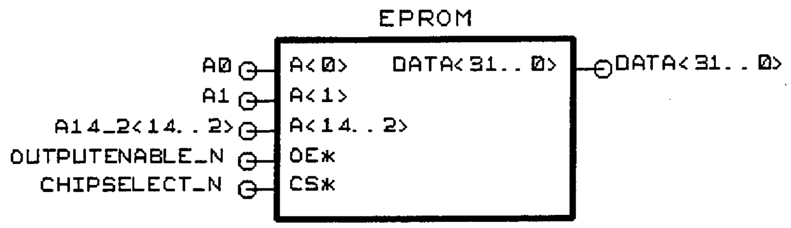


Figure 55. EPROM.

```
//*******************************************************************
//* File:  eprom.v
//*
//* Description:  Verilog behavioral file for an EPROM.
//*
//* Author:  John C. Payne, Jr.
//* Date:  10/28/98
//*******************************************************************

'timescale 1 ns /1 ps

//* Define how many entries are in the data file for internal memory
//* storage.
'define EPROM_ENTRIES  48

//*******************************************************************
//* Module:  eprom
//*
//* Description:  Verilog behavioral module for simulating an EPROM.
//*    Although because of the number of address lines, it is capable of
//*    being 128k, it has been limited to 48 entries to reduce data
//*    entry for simulation purposes.  The memory data and intialized
//*    the data file EPROM.data.
//*    This module drives the EPROM block in the Cadence Concept
//*    schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*******************************************************************
module eprom (A0, A1, A14_2, OUTPUTENABLE_N, CHIPSELECT_N, DATA);

   //* EPROM Maximum Access Times *//
   parameter
      CY27C256_max_access = 45;

   //* Module input and output lines
   input A0,
         A1;
   input [14:2] A14_2;
```

```
    input OUTPUTENABLE_N,
          CHIPSELECT_N;
    output [31:0] DATA;

    //* Internal variables (line enables)
    wire [14:0] combined_address;
    reg [31:0] memory[0:(`EPROM_ENTRIES - 1)];

    //* Intialize internal memory from data file
    initial
       begin
          $readmemh("EPROM.data",memory);
       end

    //* Combine input lines into single address
    assign combined_address[0] = A0;
    assign combined_address[1] = A1;
    assign combined_address[14:2] = A14_2;

    //* Drive data bus with data from EPROM at combined address if
    //* OUTPUTENABLE_N and CHIPSELECT_N are both low.  Drive to
    //* high impedance otherwise.
    assign #(CY27C256_max_access) DATA =
           (!OUTPUTENABLE_N &&
!CHIPSELECT_N)?memory[combined_address]:'bz;

endmodule


//***********************************************************************
//* File:  EPROM.data
//*
//* Description:  Capable of being 128K EPROM Memory File
//*               17 address lines (A[16] - A[0]) =
//*               131072 lines of 32-bit data/instructions allowed
//*    Only 48 entries have been supplied to reduce data entry for
//*    simulation purposes.
//*
//* Author:  John C. Payne, Jr.
//* Date:  10/28/98
//***********************************************************************
          //* ADDRESS
00000000  //* 00000h
00000001
00000002
00000003
00000004
00000005
00000006
00000007  //* 00007h
00000008
00000009
0000000A
0000000B
0000000C
0000000D
0000000E
0000000F  //* 0000Fh
00000010  //* 00010h
```

```
00000011
00000012
00000013
00000014
00000015
00000016
00000017  //* 00017h
00000018
00000019
0000001A
0000001B
0000001C
0000001D
0000001E
0000001F  //* 0001Fh
00000020  //* 00020h
00000021
00000022
00000023
00000024
00000025
00000026
00000027  //* 00027h
00000028
00000029
0000002A
0000002B
0000002C
0000002D
0000002E
0000002F  //* 0002Fh
```

J. SYSTEM INTERFACE

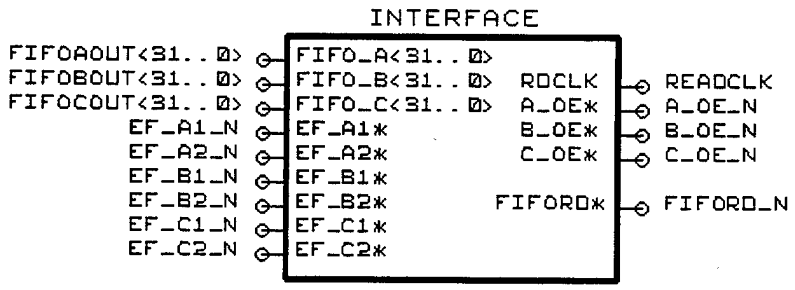


Figure 56. System Interface.

```
//*********************************************************************
//* File:  interface.v
//*
//* Description:  Verilog behavioral file for simulating the
//*               interface portion of the TMR testbed.
//*
//* Author:  John C. Payne, Jr.
//* Date:  11/15/98
//*********************************************************************

'timescale 1 ns /1 ps

'define HIGH       1
'define LOW        0


//*********************************************************************
//* Module:  interface
//*
//* Description:  Verilog behavioral module for simulating the
//*    interface of the TMR Testbed which removes the information from
//*    the three FIFOs dedicated to the three microprocessors.
//*    The data that is read from each FIFO is formatted and written to
//*    text trace file 'TMR_trace.out'.  If the file doesn't exist, it
//*    is created in the current working directory.  If the file already
//*    exists, it is emptied and overwritten.
//*    This module drives the INTERFACE block in the Cadence Concept
//*    schematic.
//*    NOTE:  Module name must match the Cadence Concept block name, but
//*    must be in lower case.  Signal names of inout, input, and output
//*    lines and size (or bus width) must match the signal names in the
//*    Cadence Concept block.
//*********************************************************************
module interface (FIFOAOUT, FIFOBOUT, FIFOCOUT, EF_A1_N, EF_A2_N,
                  EF_B1_N, EF_B2_N, EF_C1_N, EF_C2_N, READCLK,
                  A_OE_N, B_OE_N, C_OE_N, FIFORD_N);
```

```
   //* Module input and output lines
   input [31:0] FIFOAOUT,
                FIFOBOUT,
                FIFOCOUT;
   input        EF_A1_N, EF_A2_N,
                EF_B1_N, EF_B2_N,
                EF_C1_N, EF_C2_N;

   output       READCLK,
                A_OE_N,
                B_OE_N,
                C_OE_N,
                FIFORD_N;

   reg READCLK;
   wire FIFORD_N;
   wire fifoAEmpty_N, fifoBEmpty_N, fifoCEmpty_N;
   wire A_OE_N, B_OE_N, C_OE_N;
   reg [31:0] fileHandle;
   reg aOEenable, bOEenable, cOEenable, fifoRdEnable;
   reg [31:0] Adata, Bdata, Cdata, saveAdata, saveBdata, saveCdata;

   initial
   begin
      READCLK = 'LOW;
      fifoRdEnable = 'LOW;
      aOEenable = 'LOW;
      bOEenable = 'LOW;
      cOEenable = 'LOW;
      fileHandle = $fopen("TMR_trace.out");
      $fdisplay(fileHandle, "          CPU A          CPU B
CPU C");
      $fdisplay(fileHandle,
"================================================");
   end

   //* Control FIFO interface clock
   always
      #12.5 READCLK = ~READCLK;

   //* Composite FIFO empty flags.  If not empty, signals will be high.
   assign #30 fifoAEmpty_N = (EF_A1_N && EF_A2_N)?1:0;

   assign #30 fifoBEmpty_N = (EF_B1_N && EF_B2_N)?1:0;

   assign #30 fifoCEmpty_N = (EF_C1_N && EF_C2_N)?1:0;

   assign FIFORD_N = (fifoRdEnable)?0:1;

   assign A_OE_N = (aOEenable)?0:1;
   assign B_OE_N = (bOEenable)?0:1;
   assign C_OE_N = (cOEenable)?0:1;

   always
   begin
      wait((fifoAEmpty_N == 'HIGH) && (fifoBEmpty_N == 'HIGH) &&
           (fifoCEmpty_N == 'HIGH))
```

```
begin
   //* Read FIFO A -- should be address from CPU A
   @(negedge READCLK)
   begin
      #5;
      fifoRdEnable = 'HIGH;
      aOEenable = 'HIGH;
      @(posedge READCLK)
      begin
         #10;
         Adata[31:0] = FIFOAOUT[31:0];
         fifoRdEnable = 'LOW;
         aOEenable = 'LOW;
      end
   end

   //* Read FIFO B -- should be address from CPU B
   @(negedge READCLK)
   begin
      #5;
      fifoRdEnable = 'HIGH;
      bOEenable = 'HIGH;
      @(posedge READCLK)
      begin
         #10;
         Bdata[31:0] = FIFOBOUT[31:0];
         fifoRdEnable = 'LOW;
         bOEenable = 'LOW;
      end
   end

   //* Read FIFO C -- should be address from CPU C
   @(negedge READCLK)
   begin
      #5;
      fifoRdEnable = 'HIGH;
      cOEenable = 'HIGH;
      @(posedge READCLK)
      begin
         #10;
         Cdata[31:0] = FIFOCOUT[31:0];
         fifoRdEnable = 'LOW;
         cOEenable = 'LOW;
      end
   end

   //* Output address info from FIFOs to diary file
   $fdisplay(fileHandle, "Address = %h\t%h\t%h", Adata, Bdata,
             Cdata);
end

wait((fifoAEmpty_N == 'HIGH) && (fifoBEmpty_N == 'HIGH) &&
     (fifoCEmpty_N == 'HIGH))
begin

   //* Read FIFO A -- should be control from CPU A
   @(negedge READCLK)
```

```
      begin
         #5;
         fifoRdEnable = 'HIGH;
         aOEenable = 'HIGH;
         @(posedge READCLK)
         begin
            #10;
            Adata[31:0] = FIFOAOUT[31:0];
            fifoRdEnable = 'LOW;
            aOEenable = 'LOW;
         end
      end

      //* Read FIFO B -- should be control from CPU B
      @(negedge READCLK)
      begin
         #5;
         fifoRdEnable = 'HIGH;
         bOEenable = 'HIGH;
         @(posedge READCLK)
         begin
            #10;
            Bdata[31:0] = FIFOBOUT[31:0];
            fifoRdEnable = 'LOW;
            bOEenable = 'LOW;
         end
      end

      //* Read FIFO C -- should be control from CPU C
      @(negedge READCLK)
      begin
         #5;
         fifoRdEnable = 'HIGH;
         cOEenable = 'HIGH;
         @(posedge READCLK)
         begin
            #10;
            Cdata[31:0] = FIFOCOUT[31:0];
            fifoRdEnable = 'LOW;
            cOEenable = 'LOW;
         end
      end

      //* Output control info from FIFOs to diary file
      $fdisplay(fileHandle, "Control = %h\t%h\t%h", Adata, Bdata,
                Cdata);

      //* Save CONTROL data for displaying control status at end
      //* of reading DATA data from FIFO
      saveAdata = Adata;
      saveBdata = Bdata;
      saveCdata = Cdata;
   end

   wait((fifoAEmpty_N == 'HIGH) && (fifoBEmpty_N == 'HIGH) &&
        (fifoCEmpty_N == 'HIGH))
   begin
```

```
//* Read FIFO A -- should be data to/from CPU A
@(negedge READCLK)
begin
   #5;
   fifoRdEnable = 'HIGH;
   aOEenable = 'HIGH;
   @(posedge READCLK)
   begin
      #10;
      Adata[31:0] = FIFOAOUT[31:0];
      fifoRdEnable = 'LOW;
      aOEenable = 'LOW;
   end
end

//* Read FIFO B -- should be data to/from CPU B
@(negedge READCLK)
begin
   #5;
   fifoRdEnable = 'HIGH;
   bOEenable = 'HIGH;
   @(posedge READCLK)
   begin
      #10;
      Bdata[31:0] = FIFOBOUT[31:0];
      fifoRdEnable = 'LOW;
      bOEenable = 'LOW;
   end
end

//* Read FIFO C -- should be data to/from CPU C
@(negedge READCLK)
begin
   #5;
   fifoRdEnable = 'HIGH;
   cOEenable = 'HIGH;
   @(posedge READCLK)
   begin
      #10;
      Cdata[31:0] = FIFOCOUT[31:0];
      fifoRdEnable = 'LOW;
      cOEenable = 'LOW;
   end
end

//* Output data info from FIFOs to diary file
$fdisplay(fileHandle, "Data    = %h\t%h\t%h", Adata, Bdata,
          Cdata);
case(saveAdata[4:2])
   3'b010:
      $fdisplay(fileHandle, "A Control = Burst Read Word %d",
                saveAdata[1:0]);
   3'b110:
      $fdisplay(fileHandle, "A Control = Read");
   3'b101:
      $fdisplay(fileHandle, "A Control = Write");
   default:
```

```
                $fdisplay(fileHandle, "A Control = Illegal Bus
Transaction");
          endcase

          case(saveBdata[4:2])
             3'b010:
                $fdisplay(fileHandle, "B Control = Burst Read Word %d",
                          saveBdata[1:0]);
             3'b110:
                $fdisplay(fileHandle, "B Control = Read");
             3'b101:
                $fdisplay(fileHandle, "B Control = Write");
             default:
                $fdisplay(fileHandle, "B Control = Illegal Bus
Transaction");
          endcase

          case(saveCdata[4:2])
             3'b010:
                $fdisplay(fileHandle, "C Control = Burst Read Word %d",
                          saveCdata[1:0]);
             3'b110:
                $fdisplay(fileHandle, "C Control = Read");
             3'b101:
                $fdisplay(fileHandle, "C Control = Write");
             default:
                $fdisplay(fileHandle, "C Control = Illegal Bus
Transaction");
          endcase

          $fdisplay(fileHandle,
"=================================================");

       end

    end

endmodule //* end module interface
```

APPENDIX D. CADENCE SCRIPT CONTROL LANGUAGE FILES

This appendix contains two SCL files which were used to generate the simulation results obtained in Chapter V.

A. NORMAL (ERROR FREE) SCL FILE

```
//*********************************************************************
//* File:  normal.scl
//*
//* Description:  Cadence Logic Workbench Opensim Script Control
//*    Language (SCL) file.  This file executes several bus cycles for
//*    the TMR Testbed schematic.  All of the bus cycles in this file
//*    should be error free.
//*
//* Author:  John C. Payne, Jr.
//* Date:  11/30/98
//*********************************************************************

//* Definitions for transaction codes
//* (same as in verilog file for R3081 module)
NONE = 0
READ_BYTE = 1
READ_WORD = 2
READ_BURST = 3
WRITE_BYTE = 4
WRITE_WORD = 5

//* Initialize board interface lines
DEPOSIT 'PWRRESET*', 0
DEPOSIT 'TESTEN1*', 0

DEPOSIT 'FORCE_A', 0
DEPOSIT 'FORCE_B', 0
DEPOSIT 'FORCE_C', 0
DEPOSIT 'USEFIFO', 1
DEPOSIT 'PULL_UP', 1
DEPOSIT 'GND', 0

DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)

//* These initializations are necessary to prevent timing violations
//* in the simulation
DEPOSIT 'RAMCS*', 1
DEPOSIT 'EPROMCS*', 1
DEPOSIT 'INTCS*', 1
DEPOSIT 'WREN_A*', 1
DEPOSIT 'WREN_B*', 1
DEPOSIT 'WREN_C*', 1
DEPOSIT 'WREN_D*', 1
DEPOSIT 'RDEN*', 1
```

```
//* Hold board reset and release
sim 1000ns
DEPOSIT 'PWRRESET*', 1

//* Advance simulation clock during initial burst read from EPROM
//* address 1FC00000 which is initiated by the R3081 modules
while (#'VOTRD*' == 1)
   sim 25ns
while (#'VOTRD*' == 0)
   sim 25ns

sim 50ns


//*****************************************
//* Test Burst Read Bus Cycle from EPROM
//*****************************************

DEPOSIT 'A_TRANS', (READ_BURST)
DEPOSIT 'B_TRANS', (READ_BURST)
DEPOSIT 'C_TRANS', (READ_BURST)

//* Burst Read next EPROM Address
DEPOSIT 'A_ADDR', $x1FC00010
DEPOSIT 'B_ADDR', $x1FC00010
DEPOSIT 'C_ADDR', $x1FC00010

//* Advance simulation clock
while (#'VOTRD*' == 1)
   sim 25ns
while (#'VOTRD*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle completes
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz

sim 50ns


//*****************************************
//* Test Write Bus Cycle
//*****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to Lower RAM Boundary
DEPOSIT 'A_ADDR', $x00000000
DEPOSIT 'B_ADDR', $x00000000
DEPOSIT 'C_ADDR', $x00000000

DEPOSIT 'A_DATA', $x11111111
DEPOSIT 'B_DATA', $x11111111
DEPOSIT 'C_DATA', $x11111111

//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle completes
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz

sim 50ns


//*****************************************
//* Test Write Bus Cycle
//*****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x00000004
DEPOSIT 'B_ADDR', $x00000004
DEPOSIT 'C_ADDR', $x00000004

DEPOSIT 'A_DATA', $x22222222
DEPOSIT 'B_DATA', $x22222222
DEPOSIT 'C_DATA', $x22222222

while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle completes
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz

sim 50ns
```

```
//****************************************
//* Test Write Bus Cycle
//****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x00000008
DEPOSIT 'B_ADDR', $x00000008
DEPOSIT 'C_ADDR', $x00000008

DEPOSIT 'A_DATA', $x33333333
DEPOSIT 'B_DATA', $x33333333
DEPOSIT 'C_DATA', $x33333333

//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle completes
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz

sim 50ns


//****************************************
//* Test Write Bus Cycle
//****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x0000000C
DEPOSIT 'B_ADDR', $x0000000C
DEPOSIT 'C_ADDR', $x0000000C

DEPOSIT 'A_DATA', $x44444444
DEPOSIT 'B_DATA', $x44444444
DEPOSIT 'C_DATA', $x44444444

//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns
```

```
//* Advance sim clock to ensure previous cycle completes
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz

sim 50ns


//******************************************
//* Test Read Bus Cycle
//******************************************

DEPOSIT 'A_TRANS', (READ_WORD)
DEPOSIT 'B_TRANS', (READ_WORD)
DEPOSIT 'C_TRANS', (READ_WORD)

//* Read Lower RAM Boundary
DEPOSIT 'A_ADDR', $x00000000
DEPOSIT 'B_ADDR', $x00000000
DEPOSIT 'C_ADDR', $x00000000

//* Advance simulation clock
while (#'VOTRD*' == 1)
   sim 25ns
while (#'VOTRD*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle completes
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz

sim 50ns


//******************************************
//* Test Burst Read Bus Cycle from RAM
//******************************************

DEPOSIT 'A_TRANS', (READ_BURST)
DEPOSIT 'B_TRANS', (READ_BURST)
DEPOSIT 'C_TRANS', (READ_BURST)

//* Burst Read from RAM
DEPOSIT 'A_ADDR', $x00000000
DEPOSIT 'B_ADDR', $x00000000
DEPOSIT 'C_ADDR', $x00000000
```

```
//* Advance simulation clock
while (#'VOTRD*' == 1)
   sim 25ns
while (#'VOTRD*' == 0)
   sim 25ns

DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz

//* Advance sim clock to ensure previous cycle completes and FIFO is
//* emptied
sim 150ns
```

B. ERROR SCL FILE

```
//*************************************************************************
//* File:  errors.scl
//*
//* Description:  Cadence Logic Workbench Opensim Script Control
//*    Language (SCL) file.  This file executes several bus cycles for
//*    the TMR Testbed schematic.  Several of the bus cycles in this
//*    file should contain errors.
//*
//* Author:  John C. Payne, Jr.
//* Date:  11/30/98
//*************************************************************************

//* Definitions for transaction codes
//* (same as in verilog file for R3081 module)
NONE = 0
READ_BYTE = 1
READ_WORD = 2
READ_BURST = 3
WRITE_BYTE = 4
WRITE_WORD = 5

//* Initialize board interface lines
DEPOSIT 'PWRRESET*', 0
DEPOSIT 'TESTEN1*', 0

DEPOSIT 'FORCE_A', 0
DEPOSIT 'FORCE_B', 0
DEPOSIT 'FORCE_C', 0
DEPOSIT 'USEFIFO', 1
DEPOSIT 'PULL_UP', 1
DEPOSIT 'GND', 0

DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)

//* These initializations are necessary to prevent timing violations
//* in the simulation
DEPOSIT 'RAMCS*', 1
DEPOSIT 'EPROMCS*', 1
DEPOSIT 'INTCS*', 1
DEPOSIT 'WREN_A*', 1
DEPOSIT 'WREN_B*', 1
DEPOSIT 'WREN_C*', 1
DEPOSIT 'WREN_D*', 1
DEPOSIT 'RDEN*', 1

//* Hold board reset and release
sim 1000ns
DEPOSIT 'PWRRESET*', 1

//* Advance simulation clock during initial burst read from EPROM
//* address 1FC00000 which is initiated by the R3081 modules
while (#'VOTRD*' == 1)
   sim 25ns
while (#'VOTRD*' == 0)
   sim 25ns

sim 50ns


//*****************************************
//* Test Write Bus Cycle
//*    - with single error in address inputs
//*****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x00000100
DEPOSIT 'B_ADDR', $x00000000
DEPOSIT 'C_ADDR', $x00000000

DEPOSIT 'A_DATA', $x11111111
DEPOSIT 'B_DATA', $x11111111
DEPOSIT 'C_DATA', $x11111111

//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle and the interrupt
//* service routine which is initiated by the R3081 complete
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz
sim 3700ns
```

```
//*****************************************
//* Test Write Bus Cycle
//*    - with multiple errors in address
//*      inputs
//*****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x00000004
DEPOSIT 'B_ADDR', $x01000004
DEPOSIT 'C_ADDR', $x00000005

DEPOSIT 'A_DATA', $x22222222
DEPOSIT 'B_DATA', $x22222222
DEPOSIT 'C_DATA', $x22222222

//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle and the interrupt
//* service routine which is initiated by the R3081 complete
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz
sim 3700ns


//*****************************************
//* Test Write Bus Cycle
//*    - with single error in data inputs
//*****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x00000008
DEPOSIT 'B_ADDR', $x00000008
DEPOSIT 'C_ADDR', $x00000008

DEPOSIT 'A_DATA', $x33333333
DEPOSIT 'B_DATA', $x33333333
DEPOSIT 'C_DATA', $x33333337
```

```
//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle and the interrupt
//* service routine which is initiated by the R3081 complete
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz
sim 3700ns


//*****************************************
//* Test Write Bus Cycle
//*    - with multiple errors in data inputs
//*****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (WRITE_WORD)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x0000000C
DEPOSIT 'B_ADDR', $x0000000C
DEPOSIT 'C_ADDR', $x0000000C

DEPOSIT 'A_DATA', $xF4444444
DEPOSIT 'B_DATA', $x44A44444
DEPOSIT 'C_DATA', $x44444447

//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle and the interrupt
//* service routine which is initiated by the R3081 complete
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz
sim 3700ns
```

```
//*****************************************
//* Test Burst Read Bus Cycle
//*****************************************

DEPOSIT 'A_TRANS', (READ_BURST)
DEPOSIT 'B_TRANS', (READ_BURST)
DEPOSIT 'C_TRANS', (READ_BURST)

//* Burst Read from RAM
DEPOSIT 'A_ADDR', $x00000000
DEPOSIT 'B_ADDR', $x00000000
DEPOSIT 'C_ADDR', $x00000000

//* Advance simulation clock
while (#'VOTRD*' == 1)
   sim 25ns
while (#'VOTRD*' == 0)
   sim 25ns

//* Advance sim clock to ensure previous cycle completes
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz

sim 50ns


//*****************************************
//* Test Write Bus Cycle
//*    - with error in control inputs
//*****************************************

DEPOSIT 'A_TRANS', (WRITE_WORD)
DEPOSIT 'B_TRANS', (READ_BURST)
DEPOSIT 'C_TRANS', (WRITE_WORD)

//* Write to RAM
DEPOSIT 'A_ADDR', $x00004000
DEPOSIT 'B_ADDR', $x00004000
DEPOSIT 'C_ADDR', $x00004000

DEPOSIT 'A_DATA', $x78787878
DEPOSIT 'B_DATA', $x78787878
DEPOSIT 'C_DATA', $x78787878

//* Advance simulation clock
while (#'VOTWR*' == 1)
   sim 25ns
while (#'VOTWR*' == 0)
   sim 25ns
```

```
//* Advance sim clock to ensure previous cycle and the interrupt
//* service routine which is initiated by the R3081 complete
DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz
sim 3700ns

//***************************************
//* Test Read Bus Cycle
//***************************************

DEPOSIT 'A_TRANS', (READ_WORD)
DEPOSIT 'B_TRANS', (READ_WORD)
DEPOSIT 'C_TRANS', (READ_WORD)

//* Burst Read from RAM
DEPOSIT 'A_ADDR', $x00004000
DEPOSIT 'B_ADDR', $x00004000
DEPOSIT 'C_ADDR', $x00004000

//* Advance simulation clock
while (#'VOTRD*' == 1)
   sim 25ns
while (#'VOTRD*' == 0)
   sim 25ns

DEPOSIT 'A_TRANS', (NONE)
DEPOSIT 'B_TRANS', (NONE)
DEPOSIT 'C_TRANS', (NONE)
DEPOSIT 'A_ADDR', $xzzzzzzzz
DEPOSIT 'B_ADDR', $xzzzzzzzz
DEPOSIT 'C_ADDR', $xzzzzzzzz
DEPOSIT 'A_DATA', $xzzzzzzzz
DEPOSIT 'B_DATA', $xzzzzzzzz
DEPOSIT 'C_DATA', $xzzzzzzzz

//* Advance sim clock to ensure previous cycle completes and FIFO is
//* emptied
sim 150ns
```

LIST OF REFERENCES

1. Silverstein, S., "PanAmSat Scrambles to Restore Service," *Space News*, vol. 9, no. 21, p. 3, Springfield, VA, 1998.

2. Rhea, J., "The Challenges of Space on the New COTS Frontier," *Military and Aerospace Electronics*, vol. 8, no. 5, pp. 14-18, Springfield, VA, 1997.

3. McHale, John, "Space Electronics to Release Space Board Later this Year," *Military and Aerospace Electronics*, vol. 9, no. 7, p. 6, Springfield, VA, 1998.

4. Ritter, James C., "Spacecraft Anomalies and Future Trends," Radiation Effects Challenges for 21st Century Space Systems (1996 IEEE Nuclear and Space Radiation Effects Conference Short Course), IEEE Publishing Services, Piscataway, NJ, 1996.

5. *The IDT79R3071, IDT79R3081 RISController Hardware User's Manual*, Integrated Device Technology, Inc., Santa Clara, CA, 1994.

6. Johnson, Barry W., *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley, 1989.

7. Anderson, T. and P. A. Lee, *Fault Tolerance Principles and Practice*, Prentice-Hall International, 1981.

8. Wakerly, J. F., "Microcomputer Reliability Improvement Using Triple-Modular Redundancy," *Proceedings of the IEEE 64(6)*, pp. 889-895, June, 1976.

9. Ng, Andrew, *IDT79R3051 System Design Example,* RISC Microprocessor Applications Guide, pp. 1-31, Integrated Device Technology, Inc., Santa Clara, CA, 1995.

10. Thomas, Donald E. and Philip R. Moorby, *The Verilog Hardware Description Language*, 3rd Edition, Kluwer Academic Publishers, Norwell, MA, 1996.

11. R3081 Datasheet, *IDT79R3081 RISController with FPA*, file 2889.pdf, located at internet address http://www.idt.com/products/product_files/79R3081.html

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Chairman, Code EC 1
   Naval Postgraduate School
   Monterey, California 93943-5101

4. Professor Alan Ross 2
   Naval Postgraduate School
   Monterey, California 93943-5101

5. Professor Douglas Fouts 1
   Code EC/Fs
   Naval Postgraduate School
   Monterey, California 93943-5101

6. LT John C. Payne, Jr., USN 1
   6408 Chapel View Rd.
   Clifton, VA 22024

7. Ron Phelps 1
   Code SP/Ph
   Bldg. 233, Rm. 125
   Naval Postgraduate School
   Monterey, California 93943-5101